

REAL-TIME SCHEDULING FOR GPUS WITH  
APPLICATIONS IN ADVANCED AUTOMOTIVE SYSTEMS

Glenn A. Elliott

A dissertation submitted to the faculty at the University of North Carolina at Chapel Hill in partial fulfillment  
of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill  
2015

Approved by:

James H. Anderson

Sanjoy K. Baruah

Anselmo Lastra

F. Donelson Smith

Lars S. Nyland

Shinpei Kato

©2015  
Glenn A. Elliott  
ALL RIGHTS RESERVED

## **ABSTRACT**

Glenn A. Elliott: Real-Time Scheduling for GPUs with  
Applications in Advanced Automotive Systems  
(Under the direction of James H. Anderson)

Self-driving cars, once constrained to closed test tracks, are beginning to drive alongside human drivers on public roads. Loss of life or property may result if the computing systems of automated vehicles fail to respond to events at the right moment. We call such systems that must satisfy precise timing constraints “real-time systems.” Since the 1960s, researchers have developed algorithms and analytical techniques used in the development of real-time systems; however, this body of knowledge primarily applies to traditional CPU-based platforms. Unfortunately, traditional platforms cannot meet the computational requirements of self-driving cars without exceeding the power and cost constraints of commercially viable vehicles. We argue that modern graphics processing units, or GPUs, represent a feasible alternative, but new algorithms and analytical techniques must be developed in order to integrate these uniquely constrained processors into a real-time system.

The goal of the research presented in this dissertation is to discover and remedy the issues that prevent the use of GPUs in real-time systems. To overcome these issues, we design and implement a real-time multi-GPU scheduler, called GPUSync. GPUSync tightly controls access to a GPU’s computational and DMA processors, enabling simultaneous use despite potential limitations in GPU hardware. GPUSync enables tasks to migrate among GPUs, allowing new classes of real-time multi-GPU computing platforms. GPUSync employs heuristics to guide scheduling decisions to improve system efficiency without risking violations in real-time constraints. GPUSync may be paired with a wide variety of common real-time CPU schedulers. GPUSync supports closed-source GPU runtimes and drivers without loss in functionality.

We evaluate GPUSync with both analytical and runtime experiments. In our analytical experiments, we model and evaluate over fifty configurations of GPUSync. We determine which configurations support the greatest computational capacity while maintaining real-time constraints. In our runtime experiments, we

execute computer vision programs similar to those found in automated vehicles, with and without GPUSync. Our results demonstrate that GPUSync greatly reduces jitter in video processing.

Research into real-time systems with GPUs is a new area of study. Although there is prior work on such systems, no other GPU scheduling framework is as comprehensive and flexible as GPUSync.

To the McKenzies

## ACKNOWLEDGEMENTS

When I joined the computer science department at UNC, I largely believed that research as a graduate student was an independent affair: my success or failure depended entirely upon myself. I certainly would have failed had this been true. The completion of this dissertation is due in part to the knowledge, guidance, aid, and even love, from those I met and worked with during my time at Chapel Hill. I owe them all a great debt of gratitude.

First and foremost, I wish to thank my advisor, Jim Anderson, for taking a chance on a student who knew nothing of real-time systems and only thought “research on GPUs might be interesting.” On more than one occasion, Jim’s encouragement made it possible for me to complete a successful paper before a submission deadline when I had been ready to throw in the towel a day before. I would also like to thank the members of my committee, Sanjoy Baruah, Anselmo Lastra, Don Smith, Lars Nyland, and Shinpei Kato for their guidance in my research.

I also wish thank my wonderful colleagues of the Real-Time Systems Research Group at UNC. In particular, I am especially thankful for Björn Brandenburg and Bryan Ward. Björn’s tireless work on the LITMUS<sup>RT</sup> operating system, along with his techniques for real-time analysis, set the stage upon which I performed my work. The results of Bryan’s research on locking protocols (as well as Björn’s) figures heavily in the research presented herein. My work simply would not have been possible without the contributions of these two. I would also like to thank fellow LITMUS<sup>RT</sup> developers Andrea Bastoni, Jonathan Herman, and Chris Kenna for helping me work through software bugs at any hour, as well as their willingness to offer advice during the implementation of the ideas behind my research. Additionally, I wish to thank my conference paper co-authors, Jeremy Erickson, Namhoon Kim, Cong Liu, and Kecheng Yang. I thoroughly enjoyed working with each of you. Finally, I would like to thank all of the other members of the real-time systems group, including Bipasa Chattopadhyay, Zhishan Guo, Hennadiy Leontyev, Haohan Li, and Mac Mollison.

I am also grateful for the opportunities NVIDIA afforded me. My internships with the company allowed me to “peer behind the curtain” to see how GPUs and GPGPU runtimes are made. I thank Bryan Catanzaro,

Vlad Glavtchev, Mark Hairgrove, Chris Lamb, and Thierry Lepley for their guidance. I also wish to especially thank Elif Albuz and Michael Garland for giving me a free hand to explore.

I am also indebted to the staff of the UNC computer science department. I owe special thanks to Mike Stone for keeping my computing hardware going, Melissa Wood for keeping grant applications on track, and Jodie Turnbull for keeping the graduate school paperwork in order.

Finally, I wish to thank my parents and sisters for their love and support during the long years I was in North Carolina—thank you for not telling me that I was crazy for going back to school, even if you may have thought it, and even if you were right! I also want to thank Ryan and Leann McKenzie for their friendship; words cannot express how grateful I am. Lastly, I wish to thank Kim Kutz for her continued patience, love, and support, which sometimes took the form of fresh out-of-the-oven strawberry-rhubarb pie delivered to Sitterson Hall at one in the morning.

The funding for this research was provided by NSF grants CNS 0834132, CNS 0834270, CNS 1016954, CNS 1115284, CNS 1218693, CNS 1409175, CPS 1239135, and CPS 1446631; AFOSR grants FA9550-09-1-0549 and FA9550-14-1-0161; AFRL grant FA8750-11-1-0033; and ARO grants W911NF-14-1-0499 and W911NF-09-1-0535; with additional grants from General Motors, AT&T, and IBM.

## TABLE OF CONTENTS

LIST OF TABLES .....	xiv
LIST OF FIGURES .....	xvi
LIST OF ABBREVIATIONS .....	xx
Chapter 1: Introduction .....	1
1.1 Real-Time Systems .....	2
1.2 Graphics Processing Units .....	3
1.3 Real-Time GPU Applications .....	6
1.4 An Introduction to GPGPU Programming .....	7
1.4.1 GPGPU Programming .....	7
1.4.2 Real-Time GPU Scheduling .....	9
1.4.3 Real-Time Multi-GPU Scheduling .....	11
1.5 Thesis Statement .....	11
1.6 Contributions .....	12
1.6.1 A Flexible Real-Time Multi-GPU Scheduling Framework .....	12
1.6.2 Techniques for Supporting Closed-Source GPGPU Software .....	13
1.6.3 Support for Graph-Based GPGPU Applications .....	13
1.6.4 Implementation and Evaluation .....	14
1.7 Organization .....	14
Chapter 2: Background and Prior Work .....	15
2.1 Multiprocessor Real-Time Scheduling .....	15
2.1.1 Sporadic Task Model .....	15
2.1.2 Rate-Based Task Model .....	17



2.1.3	Processing Graph Method .....	18
2.1.4	Scheduling Algorithms .....	21
2.1.5	Schedulability Tests and Tardiness Bounds .....	23
2.1.6	Locking Protocols .....	27
2.1.6.1	Priority Inversions and Progress Mechanisms .....	28
2.1.6.2	Nested Locking .....	33
2.1.6.3	Priority-Inversion Blocking .....	35
2.1.7	Multiprocessor $k$ -Exclusion Locking Protocols .....	37
2.1.7.1	The $k$ -FMLP .....	38
2.1.7.2	The $R^2$ DGLP .....	41
2.1.7.3	The CK-OMLP .....	44
2.1.8	Accounting for Overheads in Schedulability Tests .....	46
2.1.8.1	Preemption-Centric Accounting .....	47
2.1.8.2	Locking Protocol Overheads .....	50
2.1.9	Integration of PI-Blocking and Overhead Accounting .....	52
2.2	Real-Time Operating Systems .....	54
2.2.1	Basic RTOS Requirements .....	54
2.2.2	LITMUS <sup>RT</sup> .....	56
2.2.3	Interrupt Handling .....	57
2.2.3.1	Linux .....	58
2.2.3.2	PREEMPT_RT .....	60
2.2.3.3	Towards Ideal Real-Time Bottom-Half Scheduling .....	62
2.3	Review of Embedded GPUs and Accelerators .....	66
2.4	GPGPU Mechanics .....	70
2.4.1	Software Architecture .....	70
2.4.2	Hardware Architecture .....	74
2.4.2.1	Execution Engine .....	76
2.4.2.2	Copy Engines .....	80

2.4.3	Other Data Transfer Mechanisms .....	81
2.4.4	Maintaining Engine Independence .....	83
2.4.5	VectorAdd Revisited .....	85
2.5	Prior Work on Accelerator Scheduling .....	87
2.5.1	Real-Time GPU Scheduling .....	87
2.5.1.1	Persistent Kernels .....	88
2.5.1.2	GPU Kernel WCET Estimation and Control .....	88
2.5.1.3	GPU Resource Scheduling .....	90
2.5.2	Real-Time DSPs and FPGA Scheduling .....	92
2.5.3	Non-Real-Time GPU Scheduling .....	94
2.5.3.1	GPU Virtualization .....	94
2.5.3.2	GPU Resource Maximization .....	96
2.5.3.3	Fair GPU Scheduling .....	97
2.6	Conclusion .....	99
Chapter 3: GPUSync .....		100
3.1	Software Architectures For GPU Schedulers .....	101
3.1.1	API-Driven vs. Command-Driven GPU Schedulers .....	101
3.1.2	Design-Space of API-Driven GPU Schedulers .....	102
3.1.2.1	GPU Scheduling in User-Space .....	103
3.1.2.2	GPU Scheduling in Kernel-Space .....	108
3.2	Design .....	114
3.2.1	Synchronization-Based Philosophy .....	114
3.2.2	System Model .....	115
3.2.3	Resource Allocation .....	115
3.2.3.1	High-Level Description .....	115
3.2.3.2	GPU Allocator .....	117
3.2.3.3	Cost Predictor .....	123

3.2.3.4	Engine Locks .....	124
3.2.4	Budget Enforcement .....	127
3.2.5	Integration .....	130
3.2.5.1	GPU Interrupt Handling .....	130
3.2.5.2	CUDA Runtime Callback Threads .....	135
3.3	Implementation .....	136
3.3.1	General Information .....	136
3.3.2	Scheduling Policies .....	137
3.3.3	Priority Propagation .....	138
3.3.4	Heuristic Plugins for the GPU Allocator .....	143
3.3.5	User Interface .....	144
3.4	Conclusion .....	145
Chapter 4:	Evaluation .....	146
4.1	Evaluation Platform .....	147
4.2	Platform Configurations .....	148
4.3	Schedulability Analysis.....	149
4.3.1	Overhead Measurement .....	149
4.3.1.1	Algorithmic Overheads .....	150
4.3.1.2	Memory Overheads .....	154
4.3.2	Scope .....	167
4.3.3	Task Model for GPU-using Sporadic Tasks.....	169
4.3.4	Blocking Analysis .....	170
4.3.4.1	Three-Phase Blocking Analysis.....	172
4.3.4.2	Coarse-Grain Blocking Analysis for Engine Locks .....	172
4.3.4.3	Detailed Blocking Analysis for Engine Locks .....	174
4.3.4.4	Detailed Blocking Analysis for the GPU Allocator .....	193
4.3.5	Overhead Accounting .....	196

4.3.5.1	Accounting for Interrupt Top-Halves .....	197
4.3.5.2	Accounting for Interrupt Bottom-Halves .....	198
4.3.5.3	Limitations .....	203
4.3.6	Schedulability Experiments .....	205
4.3.6.1	Experimental Setup .....	206
4.3.6.2	Results .....	208
4.4	Runtime Evaluation .....	221
4.4.1	Budgeting, Cost Prediction, and Affinity .....	222
4.4.1.1	Budget Performance .....	223
4.4.1.2	Cost Predictor Accuracy .....	225
4.4.1.3	Migration Frequency .....	227
4.4.2	Feature-Tracking Use-Case .....	228
4.5	Conclusion .....	234
Chapter 5: Graph Scheduling with GPUs .....		236
5.1	Motivation for Graph-Based Task Models .....	237
5.2	PGM <sup>RT</sup> .....	240
5.2.1	Graphs, Nodes, and Edges .....	240
5.2.2	Precedence Constraints and Token Transmission .....	240
5.2.3	Real-Time Concerns .....	243
5.3	OpenVX .....	244
5.4	Adding Real-Time Support to VisionWorks .....	246
5.4.1	VisionWorks and the Sporadic Task Model .....	247
5.4.2	Libgpui: An Interposed GPGPU Library for CUDA .....	252
5.4.3	VisionWorks, libgpui, and GPUSync .....	252
5.5	Evaluation of VisionWorks Under GPUSync .....	254
5.5.1	Video Stabilization .....	254
5.5.2	Experimental Setup .....	255

5.5.3	Results .....	258
5.5.3.1	Completion Delays .....	258
5.5.3.2	End-to-End Latency .....	275
5.6	Conclusion .....	280
Chapter 6: Conclusion .....		282
6.1	Summary of Results .....	282
6.2	Future Work .....	288
6.3	Closing Remarks .....	291
BIBLIOGRAPHY .....		292

## LIST OF TABLES

1.1	ADAS prototypes and related research that employ GPUs .....	7
1.2	Results of experiment reflecting unpredictable and unfair sharing of GPU resources .....	10
2.1	Summary of sporadic task set parameters .....	17
2.2	Summary parameters for describing shared resources .....	28
2.3	Summary of parameters used in preemption-centric overhead accounting .....	49
2.4	Summary of locking protocol overheads considered by preemption-centric accounting .....	51
2.5	Performance and GPGPU support of several embedded GPUs .....	67
2.6	NVIDIA software and hardware terminology with OpenCL and AMD equivalents .....	76
3.1	GPU Allocator heuristic plugin API .....	144
4.1	Summary of additional parameters to describe and analyze sporadic task sets with GPUs.....	171
4.2	All possible representative blocking chains that may delay a copy engine request .....	179
4.3	An example arrangement of copy engine requests .....	182
4.4	Summary of ILP parameters used for bounding the blocking of copy engine requests .....	188
4.5	GPUSync configuration rankings under worst-case loaded overheads .....	210
4.6	GPUSync configuration rankings under worst-case idle overheads .....	211
4.7	GPUSync configuration rankings under average-case loaded overheads .....	212
4.8	GPUSync configuration rankings under average-case idle overheads .....	213
4.9	Migration frequency under C-EDF and C-RM GPUSync configurations .....	228
5.1	Description of nodes used in the video stabilization graph of Figure 5.8 .....	256
5.2	Evaluation task set using VisionWorks' video stabilization demo .....	257
5.3	Completion delay data for SCHED_OTHER .....	262
5.4	Completion delay data for SCHED_FIFO .....	263
5.5	Completion delay data for LITMUS <sup>RT</sup> without GPUSync .....	263
5.6	Completion delay data for GPUSync for (1,6,FIFO) .....	264

5.7	Completion delay data for GPUSync for (1, 6, PRIO) .....	264
5.8	Completion delay data for GPUSync for (2, 3, FIFO) .....	265
5.9	Completion delay data for GPUSync for (2, 3, PRIO) .....	265
5.10	Completion delay data for GPUSync for (3, 2, FIFO) .....	266
5.11	Completion delay data for GPUSync for (3, 2, PRIO) .....	266
5.12	Completion delay data for GPUSync for (6, 1, FIFO) .....	267
5.13	Completion delay data for GPUSync for (6, 1, PRIO) .....	267
5.14	Average normalized completion delay data .....	268

## LIST OF FIGURES

1.1	Historical trends in CPU and GPU processor performance .....	5
1.2	Host and device code for adding two vectors in CUDA .....	8
1.3	A schedule of <code>vector_add()</code> .....	9
1.4	Matrix of high-level CPU and GPU organizational choices .....	11
2.1	Example of a PGM-specified graph .....	18
2.2	PGM-specified graph of Figure 2.1 transformed into rate-based and sporadic tasks.....	20
2.3	Task ready queues under partitioned, clustered, and global processor scheduling .....	21
2.4	Example of bounded deadline tardiness for a task set scheduled on two CPUs by G-EDF .....	25
2.5	Schedule depicting a critical section .....	27
2.6	Schedules where priority inheritance and priority boosting shorten priority inversions.....	29
2.7	Situations where multiprocessor systems require stronger progress mechanisms .....	31
2.8	Comparison of s-oblivious and s-aware pi-blocking under global scheduling .....	37
2.9	Queue structure of the k-FMLP .....	39
2.10	Queue structure of the R <sup>2</sup> DGLP .....	41
2.11	Queue structure of the CK-OMLP .....	45
2.12	Schedule depicting system overheads.....	47
2.13	Fixed-priority assignment when an I/O device is used by a single thread.....	60
2.14	Example of an unbounded priority inversion .....	61
2.15	Example of a bounded priority inversion during interrupt bottom-half processing .....	62
2.16	A pathological scenario for fixed-priority interrupt handling .....	63
2.17	A priority inversion due to the co-scheduling of a bottom-half .....	65
2.18	Layered GPGPU software architecture on Linux with closed-source drivers .....	70
2.19	Situation where a callback thread may lead to a priority inversion .....	73
2.20	High-level architecture of a modern multicore platform .....	75
2.21	Code fragments for a three-dimensional CUDA kernel .....	77
2.22	The hierarchical relation of grids, blocks, and threads .....	78



2.23	Decomposition of block threads into warps that are mapped to hardware lanes .....	79
2.24	Two streams of sequentially ordered GPU operations .....	83
2.25	Schedule of falsely dependent streamed GPU operations and corrected counterpart .....	84
2.26	A detailed schedule of host and GPU processing for the simple <code>vector_add()</code> kernel .....	86
3.1	Software architectures of API-driven GPU schedulers implemented in user-space .....	104
3.2	Software architectures of API-driven GPU schedulers implemented in kernel-space .....	110
3.3	High-level design of GPUSync's resource allocation mechanisms .....	116
3.4	A schedule of a job using GPU resources controlled by GPUSync .....	117
3.5	The structure of a GPU allocator lock .....	118
3.6	Example of budget signal handling .....	129
3.7	Architecture of GPU tasklet scheduling infrastructure using <code>klmirqd</code> .....	132
3.8	Memory layout of <code>nv_linux_state_t</code> .....	134
3.9	Relative static priorities among scheduling policies .....	138
3.10	Example of a complex chain of execution dependencies for tasks using GPUSync .....	139
3.11	Recursive algorithm to propagate changes in effective priority .....	141
4.1	Concrete examples of multicore, multi-GPU, organizational configurations .....	148
4.2	PDF of GPU top-half execution time .....	151
4.3	PDF of GPU bottom-half execution time .....	152
4.4	CCDFs of top-half and bottom-half execution times .....	153
4.5	Considered CPMD maximum overheads due to GPU traffic .....	155
4.6	Considered CPMD mean overheads due to GPU traffic .....	156
4.7	Increase in considered CPMD overheads due to GPU traffic .....	159
4.8	GPU data transmission time in an idle system .....	161
4.9	GPU data transmission time in system under load .....	162
4.10	Increase in DMA operation costs due to load .....	164
4.11	Increase in DMA operation costs due to page interleaving .....	166
4.12	Procedures for computing blocking chains for a given request scenario .....	184

4.13	Schedules with overheads due to bottom-half interrupt processing .....	199
4.14	Schedule depicting callback overheads .....	201
4.15	Illustrative ranking of configuration $\mathcal{A}$ against configuration $\mathcal{B}$ .....	208
4.16	Detailed schedulability result .....	217
4.17	Detailed result of schedulability and effective utilization .....	220
4.18	Accumulated execution engine time allocated to tasks .....	224
4.19	CDFs of percentage error in cost predictions .....	226
4.20	CDF of job response time for C-EDF with FIFO-ordered engine locks .....	231
4.21	CDF of job response time for C-EDF with priority-ordered engine locks .....	231
4.22	CDF of job response time for C-RM with priority-ordered engine locks .....	232
5.1	Dataflow graph of a simple pedestrian detector application .....	237
5.2	Transformation of a PGM-specified pedestrian detection application to sporadic tasks .....	238
5.3	Parallel execution of graph nodes .....	239
5.4	Construction of a graph in OpenVX for pedestrian detection .....	245
5.5	Procedure for PGM <sup>RT</sup> -coordinated job execution .....	249
5.6	Derivation of PGM graphs used to support the pipelined thread-per-node execution model ...	250
5.7	Overly long token critical sections may result by releasing tokens at job completion time ....	253
5.8	Dependency graph of a video stabilization application .....	255
5.9	Scenarios that lead to extreme values for measured completion delays .....	258
5.10	PDF of normalized completion delay data for SCHED_OTHER. ....	271
5.11	PDF of normalized completion delay data for SCHED_FIFO. ....	271
5.12	PDF of normalized completion delay data for LITMUS <sup>RT</sup> without GPUSync .....	271
5.13	PDF of normalized completion delay data for GPUSync with (1, 6, FIFO) .....	272
5.14	PDF of normalized completion delay data for GPUSync with (1, 6, PRIO) .....	272
5.15	PDF of normalized completion delay data for GPUSync with (2, 3, FIFO) .....	272
5.16	PDF of normalized completion delay data for GPUSync with (2, 3, PRIO) .....	272
5.17	PDF of normalized completion delay data for GPUSync with (3, 2, FIFO) .....	273

5.18	PDF of normalized completion delay data for GPUSync with (3, 2, PRIO).....	273
5.19	PDF of normalized completion delay data for GPUSync with (6, 1, FIFO) .....	273
5.20	PDF of normalized completion delay data for GPUSync with (6, 1, PRIO).....	273
5.21	CDF of normalized observed end-to-end latency .....	275
5.22	CCDF of normalized observed end-to-end latency .....	277

## LIST OF ABBREVIATIONS

ADAS	Advanced Driver Assistance Systems
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
BWI	Bandwidth Inheritance
CDF	Cumulative Distribution Function
CCDF	Complementary Cumulative Distribution Function
C-EDF	Clustered Earliest Deadline First
CE	Copy Engine
CK-OMLP	Clustered $k$ -exclusion Optimal Multiprocessor Locking Protocol
C++ AMP	C++ Accelerated Massive Parallelism
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CVA	Compliant Vector Analysis
DAG	Directed Acyclic Graph
DGL	Dynamic Group Lock
dGPU	Discrete Graphics Processing Unit
DMA	Direct Memory Access
DPCP	Distributed Priority Ceiling Protocol
DSP	Digital Signal Processor
EDF	Earliest Deadline First
EE	Execution Engine
FL	Fair-Lateness
FMLP	Flexible Multiprocessor Locking Protocol
FPGA	Field Programmable Gate Array
FPS	Frames Per Second
G-EDF	Global Earliest Deadline First
GPC	Graphics Processing Cluster
GPL	GNU General Public License

GPGPU	General-Purpose Computing on Graphics Processing Units
GPU	Graphics Processing Unit
HSA	Heterogeneous System Architecture
iGPU	Integrated Graphics Processing Unit
IORW	Input Output and Read Write
IPC	Inter-Process Communication
IPI	Inter-Processor Interrupt
ISR	Interrupt Service Routine
JLFP	Job-Level Fixed Priority
$k$ -FMLP	$k$ -exclusion Flexible Multiprocessor Locking Protocol
MPCP	Multiprocessor Priority Ceiling Protocol
MSRP	Multiprocessor Stack Resource Policy
NUMA	Non-Uniform Memory Access
O-KGLP	Optimal $k$ -exclusion Global Locking Protocol
OMLP	$\mathcal{O}(m)$ Multiprocessor Locking Protocol
OpenACC	Open Accelerators
OpenCL	Open Computing Language
OS	Operating System
PCIe	Peripheral Component Interconnect Express
PCP	Priority Ceiling Protocol
PDF	Probability Density Function
P-EDF	Partitioned Earliest Deadline First
PGM	Processing Graph Method
P2P	Peer-to-Peer
RM	Rate-Monotonic
RPC	Remote Procedure Call
RTOS	Real-Time Operating System
R <sup>2</sup> DGLP	Replica-Request Donation Global Locking Protocol
SM	Streaming Multiprocessor

SRP	Stack Resource Policy
WSS	Working Set Size

## CHAPTER 1: INTRODUCTION

Real-time systems are those that must satisfy precise timing constraints in order to meet application requirements. We often find such systems where computers sense and react to the physical world. Here, loss of life or property may result if a computer fails to act in the right moment. Real-time system designers must employ algorithms that realize predictable behavior that can be modeled by mathematical analysis. This analysis allows the designer to prove that an application's timing constraints are met. However, an algorithm can only be as predictable as allowed by the underlying software and hardware upon which it is implemented. Ensuring predictability becomes an increasing challenge as computing hardware grows in complexity. This is especially true for commodity computing platforms, which are optimized for throughput performance—often at the expense of predictability. This challenge is exemplified by the recent development of programmable graphics processing units (GPUs) that are used to perform general purpose computations. GPUs offer extraordinary performance and relative energy efficiency in comparison to traditional processors. However, today's standard GPU technology is unable to meet basic real-time predictability requirements.

The goal of the research presented in this dissertation is to discover and address the issues that prevent GPUs from being used in real-time systems. GPUs exhibit unique characteristics that are not dealt with easily using methods developed in prior real-time literature. New techniques are necessary. This dissertation presents such techniques that remove the fundamental obstacles that bar the use of GPUs in real-time systems. This research is important because it may allow GPUs to become an enabling technology for embedded real-time systems that tackle computing problems that have been outside the reach of traditional processors.

This chapter begins with a brief introduction to real-time systems. We follow with a closer look at the benefits offered by GPUs and potential real-time applications. We then discuss general purpose GPU programming at a high level, followed by the challenges to supporting real-time constraints. We then present the thesis of this dissertation and describe the dissertation's contributions. Finally, we outline the organization of this dissertation's remaining chapters.

## 1.1 Real-Time Systems

The term “real time” has different meaning in different fields. In the context of computer graphics, “real time” often equates to “real fast” or some degree of quality-of-service.<sup>1</sup> For instance, a computer graphics animation may be rendered in “real time” if image frames are generated at roughly 30 frames per second. An interactive simulation may be considered “real time” if the simulation runs at about 10 to 15 frames per second. In contrast to these throughput-oriented quality-of-service-based definitions, “real time” in the field of real-time systems is more precise. A real-time system is said to be “correct” if computations meet both logical and temporal criteria. Logical criteria require that the results of a computation must be valid. This condition is true for practically any computational system. Temporal criteria require that these results must also be made available by a designated physical time (hence, “real time”). This strict concern for temporal correctness may not necessarily be included in the aforementioned computer graphics systems where “real fast” is often good enough. Indeed, temporal correctness is as important as logical correctness in a real-time system.

A real-time workload is often embodied by a set of computational *tasks*. Each task releases recurring work, with each such release called a *job*, according to a predicable rate or time interval. The completion time of each job must satisfy some temporal constraint, such as a *deadline* that occurs within some interval of time after the job’s release. A real-time scheduler is responsible for allocating processor time to each incomplete job. A set of tasks, or *task set*, is said to be *schedulable* when timing constraints are guaranteed to always be satisfied.

A scheduling algorithm, in and of itself, does not prove schedulability. Instead, schedulability is formally proven according to an analytical *model* of the scheduling algorithm and the task set in question. These models may incorporate real-world overheads that are often a function of the scheduling algorithm, the algorithm’s implementation, and the hardware platform upon which jobs are scheduled. Overheads can have a strong effect on schedulability. As a consequence, the design of an efficient real-time system involves the *co-design* of the analytical model, the scheduling algorithm, and the algorithm’s implementation.

Over the last decade, multicore processors have become ubiquitous in computing. This has spurred interest in the design and implementation of real-time multiprocessor schedulers and analytical models. Multiprocessor schedulers can be generally classified into one of three categories: partitioned, clustered,

---

<sup>1</sup>The term “real fast” in this context is borrowed from McKenney (2009).



or global. Under partitioned scheduling, each task (and all its associated jobs) is assigned to a processor. Under global scheduling, jobs are free to migrate among all processors. Cluster scheduling is a hybrid of partitioned and global approaches: the jobs of tasks are free to migrate among an assigned subset (or cluster) of processors. Each method may be best suited to a particular application with its own temporal constraints, as there are tradeoffs among the analytical models and associated overheads for each approach.

It is reasonable to assume that many future real-time systems will use multicore processors. Support for real-time GPU computing with multicore processors is a central theme of this dissertation. Moreover, we pay special attention to the interrelations between our analytical models, scheduling algorithms, algorithm implementation, and the computing hardware.

We wish for the results of this dissertation to have bearing on practical applications, so much of the effort behind this dissertation has been on the implementation of real-time multi-GPU schedulers and their integration with real-time multiprocessor (*i.e.*, CPU) schedulers. Implementation and integration is done at the operating system (OS) level. We implement all of our solutions by extending LITMUS<sup>RT</sup>, a real-time patch to the Linux kernel (Calandrino *et al.*, 2006; Brandenburg, 2011b, 2014b). This is advantageous since we can use all Linux-based GPGPU software in our research, while also benefiting from LITMUS<sup>RT</sup>'s variety of real-time multiprocessor schedulers and its other supporting functions.

## 1.2 Graphics Processing Units

The growth of GPU technology is characterized by an evolutionary process. Early GPUs of the 1970s and 1980s were used to offload 2D rendering computations from the CPU, and support for 3D rendering was common by the end of the 1990s (Buck, 2010). With few exceptions, these GPUs were “fixed function,” meaning that rendering operations were defined *a priori* by the GPU hardware. This changed with the advent of the “programmable pipeline” in 2001. The programmable pipeline enables programmers to implement custom rendering operations called “shaders” by using program code that is executed on the GPU. Some of the early successful shader languages include the OpenGL Shading Language (GLSL) (Khronos Group, 2014b), NVIDIA’s “C for Graphics” (Cg) (NVIDIA, 2012), and Microsoft’s High-Level Shader Language (HLSL) (Microsoft, 2014).

Empowered by shader languages and programmable GPUs, researchers and developers began to exploit the generality of the programmable pipeline to solve non-graphics-related problems. This practice of using

GPUs for general-purpose computations was coined *GPGPU* by M. Harris in 2002 (Luebke *et al.*, 2004; Harris, 2009). In early GPGPU approaches, computations were expressed as shader programs, operating on graphics-oriented data (*e.g.*, pixels and vertices). Recognizing the potential of GPGPU, generalized languages and runtime environments were developed by major graphics hardware vendors and software producers to allow general purpose programs to be run on graphics hardware without the limitations imposed by graphics-focused shader languages. Notable platforms include the Compute Unified Device Architecture (CUDA) (NVIDIA, 2014c), OpenCL (Khronos Group, 2014a), and OpenACC (OpenACC, 2013). The ease of use enabled by these advances has facilitated the adoption of GPGPU in a number of fields of computing. Today, GPGPU is used to efficiently handle data-parallel compute-intensive problems such as cryptography (Harrison and Waldron, 2008), supercomputing (Meuer *et al.*, 2014), finance (Scicomp Incorporated, 2013), ray-tracing (Aila and Laine, 2009), medical imaging (Watanabe and Itagaki, 2009), video processing (Pieters *et al.*, 2009), and many others.

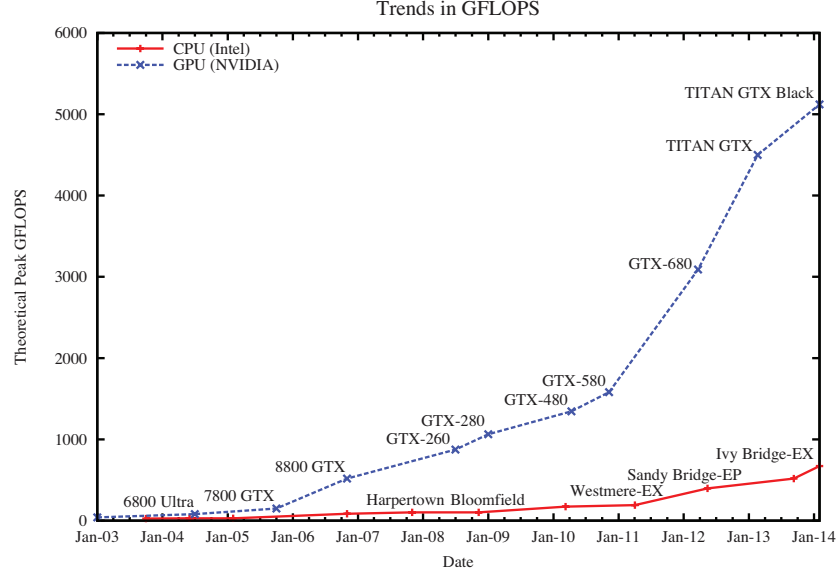
GPGPU technology has received little attention in the field of real-time systems, despite strong motivations for doing so. The strongest motivation is that the use of GPUs can significantly increase computational performance. This is illustrated in Figure 1.1(a), which depicts performance trends of high-end Intel CPUs and NVIDIA GPUs over much of the past decade (NVIDIA, 2014c; Intel, 2014; Ong *et al.*, 2010). Figure 1.1(a) plots the peak theoretical single-precision performance in terms of billions of floating point operations per second (GFLOPS). There is a clear disparity between CPU and GPU performance in favor of GPUs. For example, NVIDIA’s Titan GTX Black GPU can perform at 5,121 GFLOPS in comparison to 672 GFLOPS for the Intel Ivy Bridge-EX—the GPU has a 7.6 times greater throughput.

This disparity is even greater when we consider mobile CPUs, such as those designed by ARM. For instance, the ARM Cortex-A15 series processor has a peak theoretical performance of 8 GFLOPs at 1GHz (Rajovic *et al.*, 2014). Thus, the dual Cortex-A15 cores of Samsung’s Exynos 5250 (which runs at 1.7GHz) collectively achieve 27.2 GFLOPS. In contrast, the embedded NVIDIA K1 and PowerVR GX6650 GPUs can both achieve a reported 384 GFLOPS (Smith, 2014a), making them more than 14 times faster than the Exynos 5250.<sup>2</sup>

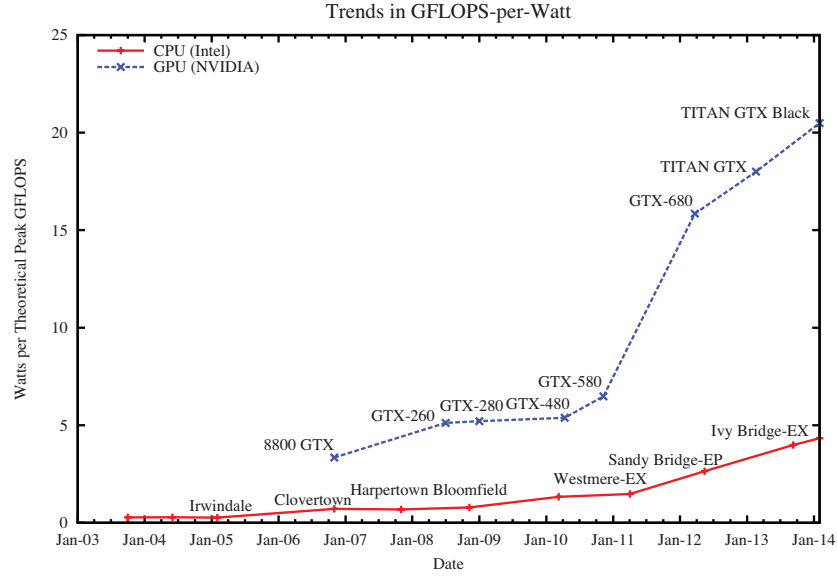
Growth in raw floating-point performance does not necessarily translate to equal gains in performance for actual applications. In the case of both CPUs and GPUs, observed theoretical performance only nears

---

<sup>2</sup>It is unfair to compare the Titan GTX Black to any embedded processor since the GPU requires a great deal more power. The Titan GTX Black consumes roughly 250 watts, while the Exynos 5250 consumes no more than 8 watts.



(a) Trend in GFLOPS.



(b) Trend in GFLOPS-per-Watt.

Figure 1.1: Historical trends in CPU and GPU processor performance.

theoretical peak performance when executing very data-parallel algorithms. However, GPU performance degrades significantly when executing inherently serial algorithms or those that contain many conditional code paths (*i.e.*, “branchy” code)—CPUs perform better in such cases (NVIDIA, 2014c). Nevertheless, a review of published research reveals that GPUs commonly increase performance over CPUs on the range of four to 20 times (Owens *et al.*, 2007) for many types of computationally heavy applications. In the context of

real-time systems, computations accelerated by GPUs may execute at higher frequencies or perform more computation per unit time, possibly improving system responsiveness or accuracy.

Power efficiency is another motivation to use GPUs in real-time systems, since real-time constraints often must be satisfied in power-constrained embedded applications. GPUs can carry out computations at a fraction of the power needed by CPUs for equivalent computations. This is illustrated in Figure 1.1(b), which depicts the GFLOPS-per-watt for the same performance points of Figure 1.1(a). Here, the Titan GTX Black can perform roughly 20.5 GFLOPS per watt in comparison to 4.3 GFLOPS per watt of the Ivy Bridge-EX—the GPU is 4.7 times more efficient. For an additional point of reference, the K1 and GX6650 integrated GPUs perform approximately 48 GFLOPS-per-watt, while the Exynos 5250 CPUs deliver approximately 6.8 GFLOPS-per-watt.<sup>3</sup>

### 1.3 Real-Time GPU Applications

There are several application domains that may benefit from real-time support for GPUs. For example, real-time-scheduled GPUs may be employed to realize predictable video compositing and encoding for use in live news and sports broadcasting (NVIDIA, 2014e). Another domain includes support of high frequency trading and other time-sensitive financial applications (King *et al.*, 2010). However, possibly the greatest potential for GPUs in real-time systems is in future automotive applications.

The domain that may benefit the most from real-time support for GPUs is in the advanced driver assistance systems (ADAS) of new and future automobiles. Here, vehicle computer systems realize “intelligent” alert and automated features that improve safety and/or the driving experience. For example, a system that alerts the driver of pedestrians in the path of the vehicle is an ADAS. An intelligent adaptive cruise control system that can automatically steer the vehicle and control its speed in stop-and-go highway traffic is another. Other ADAS features include automatic traffic sign recognition, obstacle avoidance, and driver fatigue detection. There are clear safety implications to ADAS: if the vehicle fails to act in the right moment, loss of life may result. Precise timing constraints must be met.

Common to these ADAS applications is a reliance upon a rich sensor suite. This includes video cameras, radar detectors, acoustic sensors, and lidar<sup>4</sup> sensors (Wei *et al.*, 2013). Together, these sensors generate an

---

<sup>3</sup>Power metrics for individual components of embedded processors are difficult to find. Here, we conservatively assume that the K1, GX6650, and Exynos 5250 each consume eight watts. This is a common limit for the entirety of a smartphone- and tablet-class chip, of which GPU and CPUs are merely components.

<sup>4</sup>The term “lidar” is a blend of the words “light” and “radar”.

Application	Prototypes and Research
Pedestrian, Vehicle, and Obstacle Detection	Zhang and Nevatia (2008); Wojek <i>et al.</i> (2008) Bauer <i>et al.</i> (2010); Benenson <i>et al.</i> (2012)
Traffic Sign Recognition	Mussi <i>et al.</i> (2010); Muyan-Ozcelik <i>et al.</i> (2011)
Driver Fatigue Detection	Lalonde <i>et al.</i> (2007)
Lane Following	Homm <i>et al.</i> (2010); Kuhn <i>et al.</i> (2012) Seo and Rajkumar (2014)

Table 1.1: ADAS prototypes and related research that employ GPUs.

enormous amount of data for an embedded vehicle computing system to process. It is too much in fact for traditional computing hardware to handle within a vehicle’s size, weight, and power (SWaP) constraints, much less being affordable. GPUs offer a viable alternative because many of the algorithms employed in ADAS are data parallel—ideal for GPUs. This is especially true of computer vision algorithms that operate upon video camera feeds and the point-cloud processing of LIDAR data.

Researchers have begun applying GPUs to ADAS problems, as illustrated by Table 1.1, which lists several ADAS prototypes and related research that uses GPUs. However, each prototype assumes full control of the *entire* computing system; computing resources such as CPUs and GPUs are not shared with other tasks. This does little to resolve automotive SWaP or cost constraints. If advanced automotive features are to be realistically viable, then such computations must be consolidated onto as few low-cost CPUs and GPUs as possible, while still meeting timing constraints. The design and implementation of foundational real-time methods for such systems is the focus of this dissertation research.

## 1.4 An Introduction to GPGPU Programming

A brief introduction to GPGPU programming is necessary to conceptualize the challenges of using GPUs in real-time systems. We first present a high-level description of GPGPU programming and GPU mechanics. We then discuss the limitations that prevent us from using GPUs in a real-time system without any specialized real-time mechanisms.

### 1.4.1 GPGPU Programming

Although GPUs are superior to CPUs in terms of raw performance and energy efficiency, today’s GPUs cannot operate as independent processors. Instead, GPUs used in GPGPU applications act as co-processors to CPUs. GPGPU programs are made up of a sequence of operations involving CPU code, GPU code, and, in

```

1 // Add vectors 'a' and 'b' of 'num_elements' floats and store results in 'c'
2 // using a GPU.
3 void vector_add(float *a, float *b, float *c, int num_elements) {
4     float *gpu_a, *gpu_b, *gpu_c;
5
6     ... // allocate GPU-side memory for 'gpu_a', 'gpu_b', and 'gpu_c'
7
8     // copy contents of 'a' and 'b' to corresponding buffers on the GPU
9     cudaMemcpy(gpu_a, a, num_elements*sizeof(float));
10    cudaMemcpy(gpu_b, b, num_elements*sizeof(float));
11
12    // perform 'gpu_c[i] = gpu_a[i] + gpu_b[i]' for i in [0..num_elements)
13    gpu_vector_add<<<...>>>(gpu_a, gpu_b, gpu_c, num_elements);
14
15    // copy the results of vectorAdd stored in 'gpu_c' to buffer 'c'
16    cudaMemcpy(c, gpu_c, num_elements*sizeof(float));
17
18    ... // free buffers allocated to 'gpu_a', 'gpu_b', and 'gpu_c'
19 }

```

(a) Host code for adding vectors, using a GPU.

```

1 // GPU routine for adding elements of vectors 'a' and 'b' with result stored 'c'
2 __global__
3 void gpu_vector_add(float *a, float *b, float *c, int num_elements) {
4     int i = blockDim.x * blockIdx.x + threadIdx.x;
5     if (i < num_elements) {
6         c[i] = a[i] + b[i];
7     }
8 }

```

(b) Device kernel for adding vectors.

Figure 1.2: Host and device code for adding two vectors in CUDA.

the case where GPUs are equipped with their own memory, memory copies between CPU main memory and GPU-local memory.

GPU (or “device”) code is invoked by CPU (or “host”) code, in a manner similar to a remote procedure call (RPC). Device-side procedures are commonly referred to as “kernels.”<sup>5</sup> A simple GPGPU routine (in the CUDA language) for adding the elements of two arrays (or vectors) is given in Figure 1.2. Host code appears in Figure 1.2(a) and device code in Figure 1.2(b). The routine `vector_add()` executes on the host. In lines 9 and 10 of Figure 1.2(a), memory is copied from host memory to device memory as input for the `gpu_vector_add()` kernel. In line 13, this kernel is called by the host, triggering the procedure in Figure 1.2(b) to execute on the device. The kernel’s output resides in device memory after it completes. The resulting output of the kernel is copied back to host memory on line 16 of Figure 1.2(a).

A simplified schedule for the `VectorAdd` routine is depicted in Figure 1.3. (The actual sequence of operations is more complex, as we shall see in Chapter 2, but this level of detail is sufficient for now.) From

<sup>5</sup>This unfortunate name should not be confused with an operating system kernel. A GPU kernel and operating system kernel have nothing in common.

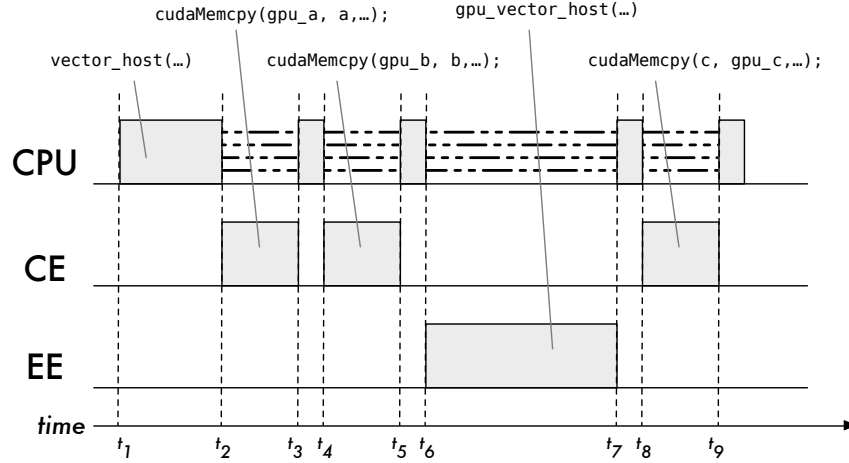


Figure 1.3: A schedule of `vector_add()`.

this schedule, we see that `vector_add()` begins execution on a CPU, starting at time  $t_1$ . The memory copy operations are carried out using direct memory access (DMA) by a “Copy Engine” (CE) processor on the time intervals  $[t_2, t_3]$ ,  $[t_4, t_5]$ , and  $[t_8, t_9]$ . The CE is a GPU component used for DMA memory operations. The kernel `gpu_vector_add()` executes on the GPU’s “Execution Engine” (EE) during the time interval  $[t_6, t_7]$ . (We discuss CEs and EEs in more depth in Chapter 2.) Observe that the CPU code waits for each GPU-related operation to complete—indicated by a set of horizontal dashed lines. During these intervals, the CPU code may suspend or spin while waiting for GPU operations to complete (the particular desired behavior may be specified by the programmer).

### 1.4.2 Real-Time GPU Scheduling

In a real-time system, CPU, CE, and EE processors must be *scheduled* according to a predictable policy to ensure that timing constraints are met. This is difficult to achieve using stock GPGPU technology. Although the routines in Figure 1.2 are simple, a great deal of complex underlying software is needed execute them. This software comes in two parts: (i) the OS device driver that manages the GPU and issues device-specific commands; and (ii) the GPGPU language runtime that interfaces with the GPU device driver to execute the GPGPU program. GPU manufacturers provide this software. However, this software is not designed with real-time requirements in mind. Worse, it is commonly distributed as closed-source, so our ability to modify the software’s behavior to support real-time requirements is constrained.

Thread	Loop Counts		
	Run 1	Run 2	Run 3
1	0	0	0
2	104	969	0
3	1,307	0	3,706
4	2,230	1,928	0
5	0	786	0
Total	3,641	3,683	3,706

Table 1.2: Reported loop counts.

Unfortunately, stock GPGPU technology offers little support for dependable scheduling policies. Contention for GPU resources is often resolved through undisclosed arbitration policies. These arbitration policies typically have no regard for task priority and may exhibit behaviors detrimental to multitasking on host CPUs. Furthermore, a single task can dominate GPU resources by issuing many or long-running operations. Allocation methods are needed that eliminate or ameliorate these problems. We demonstrate several of these points with a simple experiment.

In this experiment, we have a program that repeatedly executes the VectorAdd routine on 4,000,000-element vectors in a tight loop. We ran five instances of our program, as threads, concurrently under Linux’s general purpose scheduler (*e.g.*, a non-real-time scheduler). Our test platform has more CPUs than test threads, but all threads share the same GPU (an NVIDIA Quadro K5000). Our threads execute for a duration of 30 seconds and report the number of completed loop iterations upon completion. Table 1.2 gives the reported loop counts for three separate test runs of our experiment. We observe two important behaviors in this data. First, by comparing the values within each column, we see that each thread receives a very uneven share of GPU resources. For instance, in Run 1, Thread 2 completes 104 loops while Thread 4 completes 2,230. Second, by comparing the values within each row, we see that each thread receives a different amount of GPU resources in each experiment. For example, Thread 3 is starved in Run 2, but it receives *all* of the GPU resources in Run 3. In short, GPU resources are allocated *unfairly* and *unpredictably*.

Stock GPGPU technologies do not provide a solid foundation upon which to build a real-time system. We could endeavor to replace the closed-source software with our own, as has been explored by Kato *et al.* (2012). However, this requires a great deal of software development and reverse engineering effort. It is better to leverage existing software, if at all possible. We show in this dissertation that this is indeed possible. We



		CPU Scheduling		
		Partitioned	Clustered	Global
GPU Organization	Partitioned			
	Clustered			
	Global			

Figure 1.4: Matrix of high-level CPU and GPU organizational choices.

devise and implement mechanisms that satisfy our need for predictability while still using the manufacturer’s original software.

### 1.4.3 Real-Time Multi-GPU Scheduling

Modern hardware can support computing platforms that have multiple GPUs. Given the performance benefits GPUs offer, it is desirable to support multi-GPU computing in a real-time system. Similar to CPUs in multiprocessor scheduling, GPUs can be organized following a partitioned, clustered, or global approach. When combined with the earlier-discussed multiprocessor scheduling methods, we have nine high-level possible allocation categories, as illustrated in matrix form in Figure 1.4. Can real-time mechanisms be devised to support every configuration choice? Which configurations are best for real-time predictability? Which configurations offer the best observed real-time performance at runtime? Do configuration choices really matter? The answers to these basic questions are not immediately clear. This dissertation investigates these questions in depth.

## 1.5 Thesis Statement

GPGPU is a new technology that has received little attention in the field of real-time computing. Initial research results are promising. However, there has yet to be a comprehensive study of the topic. For instance, it is not known what tradeoffs exist among the configurations depicted in Figure 1.4. More importantly, however, it has not yet been shown to what extent a real-time system can actually benefit from GPGPU technology.

This dissertation seeks to investigate real-time GPU scheduling and demonstrate the benefits of GPGPU in real-time systems. To this end, we put forth the following thesis statement:

*The computational capacity of a real-time system can be greatly increased for data-parallel applications by the addition of GPUs as co-processors, integrated using real-time scheduling and synchronization techniques tailored to take advantage of specific GPU capabilities. Increases in computational capacity outweigh costs, both analytical and actual, introduced by management overheads and limitations of GPU hardware and software.*

## 1.6 Contributions

We now present an overview of the contributions of this dissertation that support this thesis.

### 1.6.1 A Flexible Real-Time Multi-GPU Scheduling Framework

The central contribution of this dissertation is the design of a flexible real-time multi-GPU scheduling framework. In Chapter 3, we present our framework, which is called GPUSync. We posit that GPU management is best viewed as a synchronization problem rather than one of scheduling. At its heart, GPUSync is a novel combination and adaptation of several recent advances in multiprocessor real-time synchronization made by Brandenburg *et al.* (2011) and Ward *et al.* (2012, 2013). This approach provides us established techniques for enforcing real-time predictable and an analytical framework for testing real-time schedulability.

GPUSync is highly configurable and supports every high-level CPU/GPU scheduling/organizational method depicted in Figure 1.4. For each high-level configuration, a system designer may select a low-level GPUSync configuration that best complements their analytical model or yields the best observed runtime performance (or perhaps both).

GPUSync is also designed to efficiently support practical multi-GPU scheduling, in terms of both real-time analysis and real-world performance. We employ real-time scheduling techniques that reduce pessimism in analysis. We combine online monitoring with heuristics that guide GPU scheduling decisions that improve performance while still maintaining real-time predictability. We also develop budget-enforcement techniques that allow us to mitigate the effects of GPU operations that exceed their provisioned execution times.

### 1.6.2 Techniques for Supporting Closed-Source GPGPU Software

In Chapter 2, we identify issues that may arise when we attempt to use non-real-time GPGPU software in a real-time system. Specifically, these issues relate to: **(i)** non-real-time resource arbitration, and **(ii)** the real-time scheduling of GPU device driver and GPGPU runtime computations.

We present a solution to these problems in Chapter 3. We resolve the arbitration issues of **(i)** by *wrapping* the GPGPU runtime with our own interface to GPUSync. Resource contention is already resolved when the GPGPU runtime is invoked by application code. Thus, our real-time system is no longer at the mercy of undisclosed non-real-time arbitration techniques. We address the scheduling problems of **(ii)** through *interception* techniques. For example, we intercept the OS-level interrupt processing of the GPU device driver and insert our own interrupt scheduling framework. We also intercept the creation of “helper” threads created by GPGPU runtime and assign them proper real-time scheduling priorities. These features integrate with GPUSync, allowing the framework to dynamically adjust scheduling priorities in order to maintain real-time predictability.

### 1.6.3 Support for Graph-Based GPGPU Applications

In Chapter 5, we extend real-time GPGPU support to graph-based software architectures, strengthening the relevance of GPUs in real-time systems. In such architectures, vertices represent sequential code segments that operate upon data, and edges express the flow of data among vertices. The flexibility offered by such an architecture’s inherent modularity promotes code reuse and parallel development. Also, these architectures naturally support concurrency, since parallelism can be explicitly described by the graph structure. This allows work to be scheduled in parallel and take advantage of the pipelined execution of graph processing—techniques known to improve analytical schedulability and runtime behavior.

Graph-based software architectures are well suited to efficiently handle the sensor processing and computer vision algorithms in the complex automotive applications we discussed earlier. For example, instead of repeating the same sensor processing steps within each application that uses a particular sensor, redundant computations can be consolidated in a application that distributes its results to others.

### 1.6.4 Implementation and Evaluation

In Chapter 3, we also describe the implementation of GPUSync in LITMUS<sup>RT</sup>. We discuss several of the unique implementation-related challenges we addressed, including efficient locking protocols, budget enforcement techniques, and the tracking of real-time priorities.

In Chapters 4 and 5, we evaluate our implementation in terms of analytical schedulability and runtime performance. The results of this evaluation support the central claim of this dissertation’s thesis that GPUs can be used in a real-time system, resulting in increased computational capacity.

In order to evaluate schedulability, we develop and present an analytical model of real-world system behavior under GPUSync in Chapter 4. This model incorporates carefully measured empirical overheads relating to real-time scheduling and GPU operations. Using this model, we carry out schedulability experiments in order to determine the most promising configurations of GPUSync. This evaluation is broad and is backed by data generated from tens of thousands of CPU hours of experimentation. Ultimately, we find that clustered CPU scheduling with partitioned GPUs offers the best real-time schedulability, overall. However, clustered GPU scheduling is competitive in some situations.

We demonstrate the effectiveness of our real-time GPU scheduling techniques by observing the runtime behavior of GPUSync under several synthetic and “real-world application” scenarios in Chapters 4 and 5. Among our findings, we show that although partitioned GPU scheduling may offer better real-time schedulability, clustered GPU scheduling may offer better observed real-time behavior.

## 1.7 Organization

The rest of this dissertation is organized as follows. In Chapter 2, we discuss several background topics, including the architecture and mechanics of GPUs and prior work on real-time heterogeneous multiprocessor scheduling (including GPUs). In Chapter 3, we describe the design and implementation of our configurable real-time GPU scheduling framework, GPUSync. In Chapter 4, we evaluate GPUSync in terms of theoretical schedulability and runtime experiments. In Chapter 5, we extend GPUSync to support graph-based applications and present a runtime evaluation using real-world computer vision code. We end in Chapter 6 with concluding remarks and a discussion of future work.

## CHAPTER 2: BACKGROUND AND PRIOR WORK<sup>1</sup>

In this chapter, we discuss background material and prior work on topics related to this dissertation. We begin with a discussion of real-time multiprocessor scheduling, locking protocols, and schedulability analysis. We then examine prior work on the implementation of real-time schedulers in real-time operating systems (RTOSs) and issues related to peripheral device management (specifically, device interrupt handling). We then review the current state of accelerator co-processors in the embedded domain to help motivate the timeliness of our research. We then delve into relevant aspects of GPU hardware and software architectures and programming models. Here, we also discuss the challenges of real-time GPU computing. We conclude with a review of related prior work on GPU scheduling.

### 2.1 Multiprocessor Real-Time Scheduling

We discuss several foundational elements of real-time multiprocessor scheduling in this section. We begin with a description of the analytical approach we use to model real-time workloads in this dissertation. This is followed by a discussion of the meaning of the term “schedulability” and the procedures we use for formally proving real-time correctness. We then examine the topic of resource sharing in real-time systems, and how sharing may impact schedulability analysis. Finally, we discuss the methods we use to account for real-world system overheads in schedulability analysis.

#### 2.1.1 Sporadic Task Model

A workload that is run on a real-time system is said to be *schedulable* when guarantees on timing constraints can be made. Schedulability is formally proved through analytical models. One such model is the well-studied *sporadic task model* (Mok, 1983); we focus on this task model in this research. We define the

---

<sup>1</sup> Portions of this chapter previously appeared in the proceedings of two conferences. The original citations are as follows:  
Elliott, G. and Anderson, J. (2012a). The limitations of fixed-priority interrupt handling in PREEMPT\_RT and alternative approaches. In *Proceedings of the 14th OSADL Real-Time Linux Workshop*, pages 149–155;  
Elliott, G. and Anderson, J. (2012b). Robust real-time multiprocessor interrupt handling motivated by GPUs. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 267–276.

basic elements of the sporadic task model here. Later in this section, we expand this model to incorporate resource sharing. In Chapter 4, we further expand the model to describe GPUs and GPGPU workloads.

Under the sporadic task model, we describe the computational workload as a *task set*,  $\mathcal{T}$ , that is specified as a collection of  $n$  tasks:  $\mathcal{T} \triangleq \{T_1, \dots, T_n\}$ . A *job* is a recurrent invocation of work by a task,  $T_i$ , and is denoted by  $J_{i,j}$ , where  $j$  indicates the  $j^{\text{th}}$  job of  $T_i$  (we may omit the subscript  $j$  if the particular job invocation is inconsequential). Task  $T_i$  is described by a tuple of three parameters:  $(e_i, p_i, d_i)$ . The worst-case execution time (WCET) of a job is given by  $e_i$ . The releases of jobs  $J_{i,j}$  and  $J_{i,j+1}$  have a minimum release separation time described by the task's *period*,  $p_i$ . A task is said to be *periodic* (instead of sporadic) if its jobs are always separated by  $p_i$  time units. Job  $J_{i,j}$  is released (*arrives*) at time  $a_{i,j}$  and completes (*finishes*) at time  $f_{i,j}$ . Each job has a *precedence constraint*: although job  $J_{i,j+1}$  may be released before job  $J_{i,j}$  completes,  $J_{i,j+1}$  cannot be scheduled until after  $f_{i,j}$ . A *pending* job is an incomplete released job that has had its precedence constraint met. The *response time* of  $J_{i,j}$  is

$$r_{i,j} \triangleq f_{i,j} - a_{i,j}. \quad (2.1)$$

Every task has a *relative deadline*,  $d_i$ . A task is said to have an implicit, constrained, or arbitrary deadline if  $d_i = p_i$ ,  $d_i \leq p_i$ , or  $d_i \geq 0$ , respectively. Every job has an *absolute deadline*, defined by

$$D_{i,j} \triangleq a_{i,j} + d_i. \quad (2.2)$$

Each job must execute for at most  $e_i$  time units in order to complete, and it must receive this execution time by  $D_{i,j}$  to meet its deadline. We define *lateness* by

$$l_{i,j} \triangleq f_{i,j} - D_{i,j}. \quad (2.3)$$

Deadline *tardiness* is defined by

$$x_{i,j} \triangleq \max(0, l_{i,j}). \quad (2.4)$$

The *utilization* of a task quantifies the long-term processor share required by the task and is given by the term

$$u_i \triangleq \frac{e_i}{p_i}. \quad (2.5)$$

Task Set and Scheduler Parameters	
$m$	number of CPUs
$c$	CPU cluster size
$\mathcal{T}$	task set
$n$	number of tasks in task set
$U(\mathcal{T})$	task set utilization
Parameters of Task $T_i$	
$e_i$	job worst-case execution time
$p_i$	period
$d_i$	relative deadline
$u_i$	utilization
Parameters of Job $J_{i,j}$	
$a_{i,j}$	release (arrival) time
$f_{i,j}$	completion (finish) time
$D_{i,j}$	absolute deadline
$r_{i,j}$	response time
$l_{i,j}$	deadline lateness (may be negative)
$x_{i,j}$	deadline tardiness

Table 2.1: Summary of sporadic task set parameters.

The total utilization of a task set is given by

$$U(\mathcal{T}) = \sum_{i=1}^n u_i. \quad (2.6)$$

Table 2.1 summarizes the various parameters we use in modeling a sporadic task set.

### 2.1.2 Rate-Based Task Model

The *rate-based task model* provides a similar approach to describing the workload of a real-time system as the sporadic task model (Jeffay and Goddard, 1999). Instead of using  $p_i$  to describe the arrival sequence of task  $T_i$ 's jobs, we use  $\chi_i$  to specify the maximum *number* of jobs of task  $T_i$  that may arrive within a time window of  $v_i$  time units.<sup>2</sup> Thus, each task is described by a tuple of four parameters:  $(e_i, \chi_i, v_i, d_i)$ . We reuse all of the sporadic task model parameters we described earlier, except for  $p_i$  and  $u_i$ . Utilization is given by

$$u_i^{rb} \triangleq e_i \cdot \frac{\chi_i}{v_i}. \quad (2.7)$$

---

<sup>2</sup> Jeffay and Goddard use the symbols  $x_i$  and  $y_i$  instead of  $\chi_i$  and  $v_i$ , respectively. We deviate from their notation to avoid confusion with other parameters we define. For example, we use  $x_i$  to denote deadline tardiness.

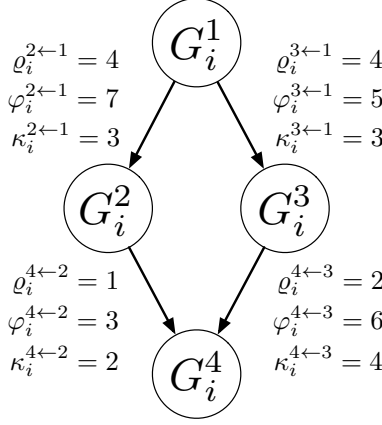


Figure 2.1: Example of a PGM-specified graph (courtesy of Liu and Anderson (2010)).

We do not use the rate-based task model directly to determine schedulability in this dissertation. However, we do use it as an *intermediary* representation in the process of transforming a more complicated real-time model into the sporadic task model. We discuss this next.

### 2.1.3 Processing Graph Method

The sporadic and rate-based task models are limited in that they describe a set of independent tasks. However, real-time workloads are not always so simple. For example, the sporadic and rate-based task models lack the necessary expressiveness to describe the scenario where the input of a job of one task  $T_i$  is dependent upon the output of a job of another task  $T_j$ . We see this type of inter-task dependence in graph-based software architectures. The Processing Graph Method (PGM) is an expressive model for describing such software architectures.

PGM describes the dependencies among jobs in terms of producer/consumer relationships. The workload is described by set of  $n$  graphs:  $\mathcal{G} \triangleq \{G_1, \dots, G_n\}$ . Each graph is comprised of subtasks,  $G_i \triangleq \{G_i^1, \dots, G_i^{z_i}\}$ , where  $z_i$  denotes the number of subtasks in  $G_i$ . Figure 2.1 depicts a graph expressed in PGM. We represent each subtask with a node. A directed edge connecting a *producer* subtask to a dependent *consumer* subtask reflects the data dependencies between connected nodes. The production and consumption of data is modeled by *tokens*, where producers produce tokens and consumers consume them. Each edge is described by three parameters that describe token production and consumption. The number of tokens produced by a subtask  $G_i^j$  for subtask  $G_i^k$  each time a job of  $G_i^j$  completes is denoted by  $\rho_i^{k \leftarrow j}$ . The number of tokens consumed by each job of  $G_i^k$  is given by  $\kappa_i^{k \leftarrow j}$ . Finally, a threshold on the number of tokens that must be available on the



edge connecting  $G_i^j$  and  $G_i^k$  before a job of  $G_i^k$  may execute is given by  $\phi_i^{k \leftarrow j}$ . We denote the set of subtasks that directly generate input for  $G_i^j$  with the function  $pred(G_i^j)$ . Likewise, we denote the set of subtasks that directly consume the output of  $G_i^j$  with the function  $cons(G_i^j)$ . We attach rate-based arrival parameters to the source nodes of each graph. Similar to the rate-based task model, we use  $\chi_i^j$  to specify the maximum *number* of jobs of task  $G_i^j$  that may arrive within a time window of  $v_i^j$  time units.

Goddard presents a procedure for transforming a set of PGM graphs into a rate-based task set in his Ph.D. dissertation (Goddard, 1998). Building upon this work, Liu and Anderson developed a method to transform the PGM-derived rate-based task set into a sporadic task set, provided that the underlying graph contains no cycles (*i.e.*, a directed acyclic graph (DAG)) (Liu and Anderson, 2010).

We now present the procedure for transforming a PGM-based graph into a sporadic task set, via an intermediate transformation into a rate-based task set. We direct the reader to Goddard (1998) and Liu (2013) for the justifications behind the graph-to-rate-based-task-set and rate-based-to-sporadic-task-set transformations, respectively.

We first assume that rate-based arrival parameters have been assigned to all source nodes of every graph. We then derive these parameters for the remaining nodes using the equations

$$v_i^k \triangleq lcm \left\{ \frac{\kappa_i^{k \leftarrow v} \cdot v_i^v}{gcd(\phi_i^{k \leftarrow v} \cdot \chi_i^v, \kappa_i^{k \leftarrow v})} \mid v \in pred(G_i^k) \right\}, \quad (2.8)$$

$$\chi_i^k \triangleq v_i^k \cdot \frac{\phi_i^{k \leftarrow v}}{\kappa_i^{k \leftarrow v}} \cdot \frac{\chi_i^v}{v_i^v}, \text{ where } G_i^v \in pred(G_i^k). \quad (2.9)$$

We compute a relative deadline for every task using the equation:

$$d_i^k \triangleq \frac{v_i^k}{\chi_i^k}. \quad (2.10)$$

At the end of this transformation, we have a rate-based task set,  $\mathcal{T}^{rb}$ , derived from  $\mathcal{G}$ . We transform  $\mathcal{T}^{rb}$  into a sporadic task set by replacing the terms  $\chi_i^k$  and  $v_i^k$  of each task with a period equal to each task's relative deadline:

$$p_i^k \triangleq \frac{v_i^k}{\chi_i^k}. \quad (2.11)$$

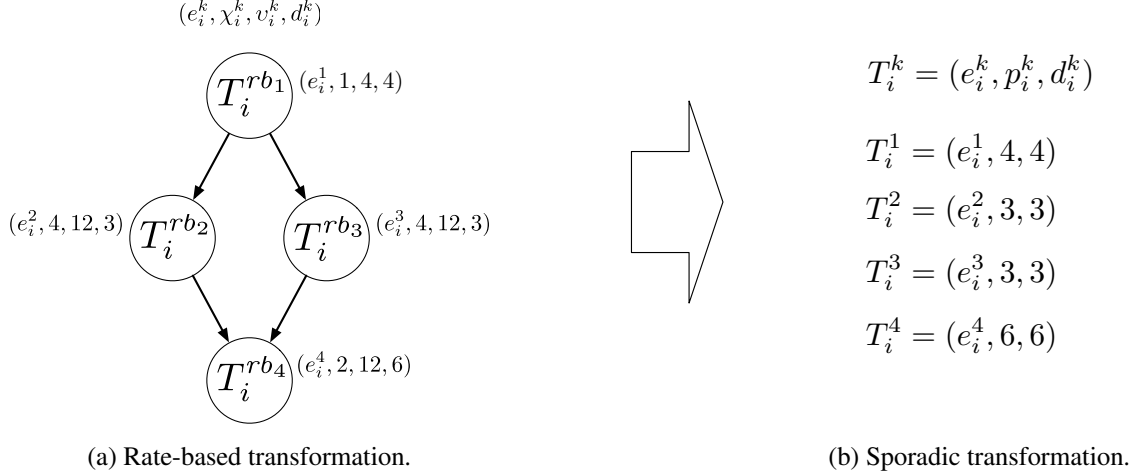


Figure 2.2: PGM-specified graph of Figure 2.1 transformed into rate-based (a) and sporadic (b) tasks.

**Example 2.1.** Consider the PGM-specified graph in Figure 2.1. Let us assume that  $G_i^1$  has an execution rate of  $(\chi_i^1 = 1, v_i^1 = 4)$ .<sup>3</sup> We now transform the graph  $G_i$  into a set of rate-based tasks, followed by a transformation into sporadic tasks. These transformations are illustrated in Figure 2.2.

We apply Equations (2.8) and (2.9) to find  $v_i^2 = lcm\left\{\frac{3 \cdot 4}{gcd(4 \cdot 1, 3)}\right\} = 12$  and  $\chi_i^2 = 12 \cdot \frac{4 \cdot 1}{3 \cdot 4} = 4$  for the rate-based task  $T_i^{rb2}$ . We use Equation (2.10) to find that the relative deadline for  $T_i^{rb2}$  is  $d_i^2 = \frac{12}{4} = 3$ . The rate-based task  $T_i^{rb3}$  is similarly defined, since  $\kappa_i^{2 \leftarrow 1} = \kappa_i^{3 \leftarrow 1}$  and  $\varrho_i^{2 \leftarrow 1} = \varrho_i^{3 \leftarrow 1}$ . Also,  $v_i^4 = lcm\left\{\frac{2 \cdot 12}{gcd(4, 2)}, \frac{4 \cdot 12}{gcd(8, 4)}\right\} = lcm\{12, 12\} = 12$  and  $\chi_i^4 = 12 \cdot \frac{1}{2} \cdot \frac{4}{12} = 2$  for the rate-based task  $T_i^{rb4}$ . The relative deadline is  $d_i^4 = \frac{12}{2} = 6$ . To transform the rate-based tasks to sporadic tasks, we merely take the rate-based relative deadline as the period for each task.  $\diamond$

Although we model the derived sporadic task set as a set of independent tasks, a scheduler must take steps, *at runtime*, to **(i)** efficiently track the production and consumption of tokens on each edge; and **(ii)** dynamically adjust the release time of jobs to ensure token input constraints are satisfied. We may accomplish **(i)** by enabling token-producing jobs to notify the scheduler of when token constraints are satisfied. Liu and Anderson (2010) specify how to address **(ii)**: we delay the release of any sporadic job until inputs are satisfied. Suppose a job  $J_{i,j}^k$  is “released” at time  $a_{i,j}^k$  as a conventional sporadic task, but that the token constraints of  $J_{i,j}^k$  are not satisfied until a later time,  $t$ . In such a case, we adjust the release time of job  $J_{i,j}^k$  to  $a_{i,j}^k = t$  and its deadline to  $D_{i,j}^k = t + d_i^k$ . We present an efficient implementation for token constraint tracking and release-time adjustment in Chapter 5.

<sup>3</sup>This example is courtesy of Liu and Anderson (2010).

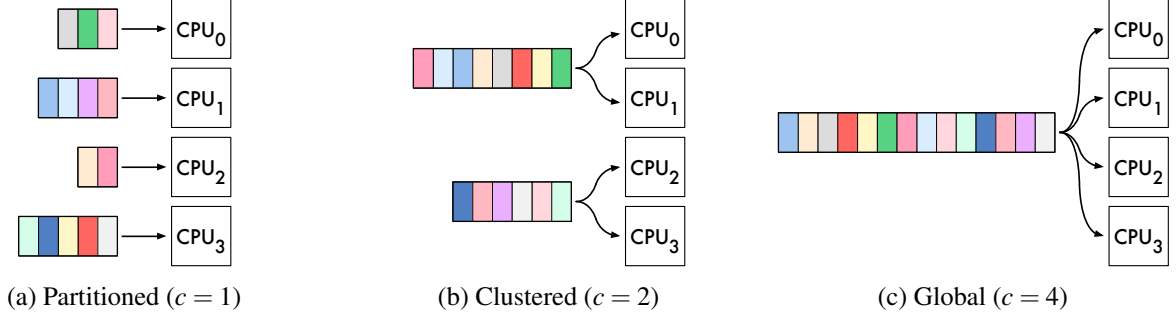


Figure 2.3: Each processor cluster is fed by a dedicated ready queue of jobs. Example with  $m = 4$ .

#### 2.1.4 Scheduling Algorithms

We now discuss the algorithms we use to schedule a given sporadic task set on set of processors, as well as the formal analysis we use to determine whether timing constraints can be met.

The task set  $\mathcal{T}$  is scheduled on hardware platform consisting of  $m$  processors (or cores). These processors may be divided into disjoint clusters of  $c$  processors each. Each task (and all its associated jobs) is assigned to a cluster. When  $c = 1$ ,  $1 < c < m$ , or  $c = m$ , the multiprocessor system is said to be scheduled by a partitioned, clustered, or global scheduler, respectively.

An incomplete released job is *ready* if it is available for execution, it is *scheduled* if the job is executing on a processor, and it is *suspended* if the job cannot be scheduled (for whatever reason). A scheduled job is either *preemptible* or *non-preemptible*, and cannot be descheduled while it is non-preemptible.

Each processor cluster draws ready jobs from a dedicated priority-ordered ready queue. This is depicted in Figure 2.3 for a system with four processors under partitioned (inset (a)), clustered (inset (b)), and global (inset (c)) scheduling. In this figure, each ready job is depicted by a different shaded box. Jobs enter the ready queue when they are released, resume from a suspension, or are preempted. Jobs exit the ready queue when they are scheduled. A job is free to migrate among all processors within its assigned cluster. Of course, no migration is possible under partitioned scheduling, since each cluster is made up of only one processor.

The research in this dissertation focuses on multiprocessor scheduling algorithms; specifically, *job-level fixed-priority* (JLFP) scheduling algorithms where a fixed scheduling priority is assigned to a job when the job is released. We further classify JLFP scheduling algorithms as *fixed-* or *dynamic-priority*. Fixed-priority (FP) scheduling algorithms assign the same priority to all jobs of the same task, though each task may have a different priority. Dynamic-priority scheduling algorithms assign a priority to a job upon release—every job

of every task may have a different priority. The rate-monotonic (RM) scheduler prioritizes tasks with shorter periods over those with longer ones. Similarly, the deadline-monotonic (DM) scheduler prioritizes tasks according to relative deadlines instead of period. Period and relative deadline parameters are constant, so RM and DM are FP schedulers. The earliest-deadline-first (EDF) scheduler prioritizes jobs with earlier *absolute* deadlines over those with later ones. EDF is a dynamic-priority scheduler since the absolute deadline depends upon the release time of a job. Similarly, the recently developed “fair-lateness” (FL) scheduler prioritizes jobs using pseudo-deadlines called *priority points* (Erickson, 2014). The relative priority point of a task, denoted by the parameter  $y_i$ , is determined by the following formula:

$$y_i \triangleq d_i - \frac{m-1}{m} e_i. \quad (2.12)$$

Similarly, an absolute priority point of a job, denoted by the parameter  $Y_{i,j}$ , is given by:

$$Y_{i,j} \triangleq D_{i,j} - \frac{m-1}{m} e_i. \quad (2.13)$$

FL is also a dynamic-priority scheduler.

We combine the processor cluster organization with a prioritization scheme to create a multiprocessor scheduling algorithm. For example, we refer to the multiprocessor scheduler defined by combination of partitioned processor organization ( $c = 1$ ) with EDF prioritization as “partitioned EDF” (P-EDF). Likewise, we get the “global RM” (G-RM) scheduler by combining globally organized processors ( $c = m$ ) with RM prioritization.

Clustered approaches are particularly effective on large multiprocessor systems where migration costs are high due to high interprocessor communication costs. Migration costs are reduced if processors that can communicate efficiently, such as through shared caches, are clustered together. *Semi-partitioned* scheduling is another hybrid approach whereby most tasks are partitioned to a individual processors, while remaining tasks may migrate between two or more processors (Anderson *et al.*, 2005). Semi-partitioned algorithms are also effective since migration costs are eliminated for most tasks. We will not discuss semi-partitioned algorithms any further; we only mention them here for the sake of completeness.

### 2.1.5 Schedulability Tests and Tardiness Bounds

A schedule is *feasible* for a task set if all timing constraints are met. A task set is *schedulable* under a given scheduling algorithm if the algorithm always generates a feasible schedule. (A scheduling algorithm is *optimal* if it always generates a feasible schedule, if one exists.) Our definitions of feasibility and schedulability are with respect to some notion of “timing constraints”—these may be application-specific. We call systems where *all* deadlines must be met *hard real-time* (HRT) systems. Applications with HRT requirements are found in safety-critical applications where loss of life or damage may occur if a deadline is missed. We call systems where some deadline misses are acceptable *soft real-time* (SRT) systems. A video decoder is an example of an application with SRT requirements. This definition of SRT remains general and can be further refined. In the context of this dissertation, an SRT system is one where deadline tardiness (the margin by which a deadline may be missed) is *bounded*.

A *schedulability test* is a procedure that determines if a given task set is schedulable. For instance, a classic result from a seminal work by Liu and Layland (1973) states that any periodic task set scheduled by uniprocessor RM scheduling is HRT-schedulable if

$$U(\mathcal{T}) \leq n(2^{\frac{1}{n}} - 1). \quad (2.14)$$

This schedulability test is only *sufficient*, as some task sets *may* be schedulable with utilizations greater than  $n(2^{\frac{1}{n}} - 1)$ . In the same work, Liu and Layland also showed that any periodic task set scheduled under uniprocessor EDF scheduling is HRT-schedulable iff

$$U(\mathcal{T}) \leq 1. \quad (2.15)$$

Since any task set with a utilization *greater* than one has no feasible schedule, uniprocessor EDF is an optimal scheduler.

The development of schedulability tests has been, and continues to be, a central topic in real-time systems research. This research has resulted in numerous schedulability tests—each may evaluate schedulability under a different set of timing constraints, scheduling algorithms, and assumptions of task set characteristics. The primary schedulability test we concern ourselves with in this dissertation is the SRT test developed by Devi and Anderson (2006) for implicit-deadline sporadic task sets scheduled by G-EDF: any conventional

sporadic task set is schedulable with bounded deadline tardiness if the constraints

$$U(\mathcal{T}) \leq m \quad (2.16)$$

and

$$\forall T_i : \quad u_i \leq 1 \quad (2.17)$$

hold true. Inequality (2.16) defines the task set utilization constraint, and Inequality (2.17) defines a per-task utilization constraint. These constraints can also be used to evaluate the schedulability (with bounded tardiness) of task sets with arbitrary deadlines (Erickson, 2014).

We can compute deadline tardiness bounds for schedulable implicit-deadline task sets. We define several terms and functions in order to describe this process. Let  $e_{\min}$  be the smallest job execution time among the tasks in  $\mathcal{T}$ . We use  $\mathcal{E}(\mathcal{T})$  to define an operation that returns the subset of  $m - 1$  tasks in  $\mathcal{T}$  with the largest  $e_j$ . Similarly, we use  $\mathcal{U}(\mathcal{T})$  to define an operation that returns the subset of  $m - 2$  tasks in  $\mathcal{T}$  with the largest  $u_k$ . The deadline tardiness of any job is bounded by

$$x_i = e_i + \frac{(\sum_{T_j \in \mathcal{E}(\mathcal{T})} e_j) - e_{\min}}{m - \sum_{T_k \in \mathcal{U}(\mathcal{T})} u_k}. \quad (2.18)$$

We may extend the above test and tardiness bounds to C-EDF by testing each cluster individually.

In cases where  $\lceil U(\mathcal{T}) \rceil < m$ , we may obtain tighter tardiness bounds by substituting the term  $m$  with  $\hat{m}$ , where

$$\hat{m} \triangleq \lceil U(\mathcal{T}) \rceil, \quad (2.19)$$

in Equation (2.18) and the definitions of  $\mathcal{E}(\mathcal{T})$  and  $\mathcal{U}(\mathcal{T})$ . This optimization reflects the observation that a task set only requires  $\lceil U(\mathcal{T}) \rceil$  processors to be schedulable with bounded tardiness. The bound provided by Equation (2.18) may be tightened if we assume *fewer* than  $m$  processors in analysis.

**Example 2.2.** Figure 2.4 depicts the schedule for three periodic implicit-deadline tasks scheduled under G-EDF on two processors ( $m = 2$ ), with deadline ties broken by task index. The tasks share the same parameters:  $T_i = (e_i = 8, p_i = 12, d_i = 12)$ . After time  $t = 12$ , the schedule settles into a steady pattern.

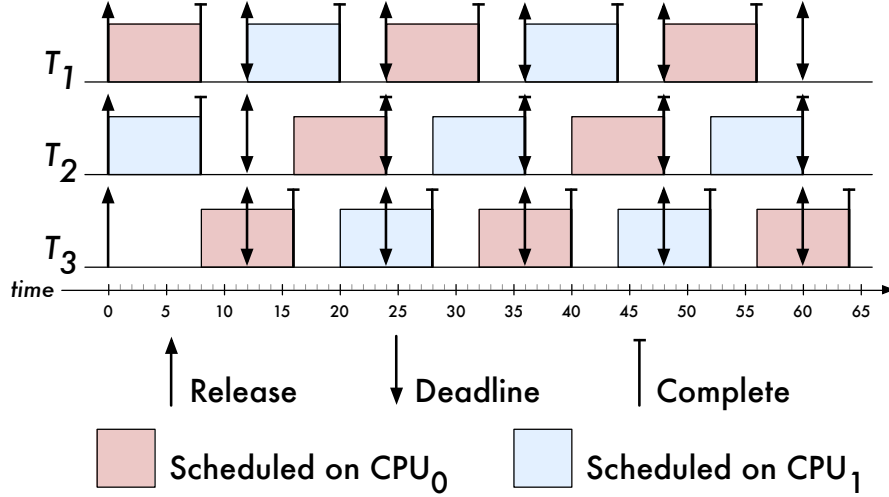


Figure 2.4: Example of bounded deadline tardiness for a task set scheduled under G-EDF on two CPUs.

This task set satisfies the task set utilization (Inequality (2.16)) and per-task utilization (Inequality (2.17)) constraints since  $U(\mathcal{T}) = 2$  and  $u_i = \frac{2}{3}$ . Using Equation (2.18) to analytically bound deadline tardiness, we find that no job will miss its deadline by more than eight time units.  $\diamond$

In Figure 2.4, we see intuitively that task  $T_3$  will never miss its deadline by more than four time units. In contrast, Equation (2.18) bounds tardiness by eight time units. This difference of four time units is evidence of pessimism in analysis. Compliant Vector Analysis (CVA) by Erickson (2014) offers tighter tardiness bounds. CVA computes bounds on deadline *lateness*, instead of tardiness, by solving a linear program. We present the CVA linear program here, but we direct the reader to Erickson (2014) for justification.

CVA uses pseudo-deadline priority points, which we discussed with respect to FL-scheduling in Section 2.1.4, and also defines several additional terms. Under CVA analysis of EDF scheduling, the relative priority point and deadline of a task coincide (*i.e.*,  $y_i = d_i$ ). We characterize processor demand by task  $T_i$  with the function

$$S_i(y_i) = e_i \cdot \max \left\{ 0, 1 - \frac{y_i}{p_i} \right\}. \quad (2.20)$$

Total demand is given by

$$S(\vec{y}) = \sum_{T_i \in \mathcal{T}} S_i(y_i). \quad (2.21)$$

Response time and lateness bounds are defined recursively, with  $\hat{x}_i$  as a real value:

$$r_i = y_i + \hat{x}_i + e_i, \quad (2.22)$$

and

$$l_i = y_i + \hat{x}_i + e_i - d_i. \quad (2.23)$$

The function

$$G(\vec{\hat{x}}, \vec{y}) = \sum_{\hat{m}-1 \text{ largest}} (\hat{x}_i u_i + e_i - S_i(y_i)) \quad (2.24)$$

denotes the processor demand from tasks that can contribute to job lateness.<sup>4</sup> Finally, let

$$s = G(\vec{\hat{x}}, \vec{y}) + S(\vec{y}). \quad (2.25)$$

We define additional variables  $S_i$ ,  $S_{\text{sum}}$ ,  $G$ ,  $b$ , and  $z_i$  for the linear program. We find values for  $\hat{x}_i$  by solving the following linear program:

$$\begin{aligned} \text{Minimize: } & s \\ \text{Subject to: } & \hat{x}_i = \frac{s - e_i}{m} & \forall i, \\ & S_i \geq 0 & \forall i, \\ & S_i \geq e_i \cdot \left(1 - \frac{y_i}{p_i}\right) & \forall i, \\ & z_i \geq 0 & \forall i, \\ & z_i \geq \hat{x}_i u_i + e_i - S_i - b & \forall i, \\ & S_{\text{sum}} = \sum_{T_i \in T} S_i, \\ & G = b \cdot (\hat{m} - 1) + \sum_{T_i \in T} z_i, \\ & s \geq G + S_{\text{sum}} \end{aligned}$$

With values for  $\hat{x}_i$  from the solution to the linear program, we compute bounds for  $l_i$  using Equation (2.23).

There are three main advantages to CVA over Devi's method (Equation (2.18)). First, CVA usually gives tighter bounds than Equation (2.18).<sup>5</sup> Second, CVA computes lateness instead of tardiness. In some instances, CVA may compute a *negative* value for  $l_i$ , indicating that a job of task  $T_i$  never misses a deadline. Finally,

---

<sup>4</sup>Observe that we use the term  $\hat{m}$  defined by Equation (2.19).

<sup>5</sup> CVA strictly dominates Equation (2.18) with additional enhancements to its linear program (Erickson, 2014).



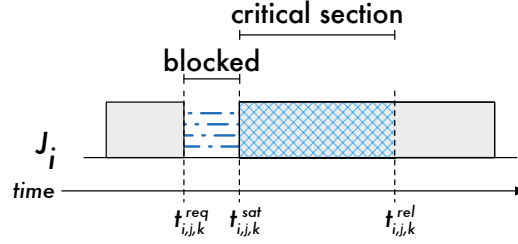


Figure 2.5: Critical section of job  $J_i$ .

CVA can be applied to task sets that include tasks with arbitrary deadlines, while Equation (2.18) may only be applied task sets made up entirely of tasks with implicit deadlines.

### 2.1.6 Locking Protocols

The sporadic task model can be extended to allow a set of serially-reusable shared resources (such as shared data objects and I/O devices) to be specified. Access to these resources must be *mutually exclusive*, in that only one job may access the resource at a time. We denote  $q$  such resources by  $\ell_1, \dots, \ell_q$ . When a job  $J_i$  requires a resource  $\ell_j$ , it issues a *request*  $R_{i,j,k}$  for  $\ell_j$  at time  $t_{i,j,k}^{req}$ . The subscript  $k$  denotes the  $k^{th}$  non-nested request of  $J_i$  for  $\ell_j$ .  $R_{i,j,k}$  is *satisfied* as soon as  $J_i$  holds  $\ell_j$  at time  $t_{i,j,k}^{sat}$ , and completes when  $J_i$  *releases*  $\ell_j$  at time  $t_{i,j,k}^{rel}$ . This sequence of events is illustrated by Figure 2.5. We call the computation and operations (such as I/O) of  $J_i$  performed within the interval  $[t_{i,j,k}^{sat}, t_{i,j,k}^{rel}]$  a *critical section*. The *length* of a critical section of job  $J_i$  for request  $R_{i,j,k}$  of resource  $\ell_j$  given by

$$L_{i,j,k} \triangleq t_{i,j,k}^{rel} - t_{i,j,k}^{sat} \quad (2.26)$$

when job  $J_i$  executes without preemption. We denote the longest critical section for resource  $\ell_j$  of any job of  $T_i$  by  $L_{i,j}^{max}$ . The longest critical section of resource  $\ell_j$  of any task is given by

$$L_j^{max} \triangleq \max_{T_i \in \mathcal{T}} \{L_{i,j}^{max}\}. \quad (2.27)$$

We denote the number of times a job  $J_i$  may issue a request for resource  $\ell_j$  with  $\eta_{i,j}$ . Finally, the total number of resource requests of *all* resources requested by  $J_i$  is denoted by  $\eta_i$ . The above parameters are summarized in Table 2.2.

Parameter	Meaning
$\ell_j$	$j^{th}$ serially reusable shared resource
$R_{i,j,k}$	$k^{th}$ request of job $J_i$ for resource $\ell_j$
$L_{i,j,k}$	critical section length of request $R_{i,j,k}$
$L_{i,j}^{max}$	longest critical section of any request from a job of $T_i$ for resource $\ell_j$
$L_j^{max}$	longest critical section of any request for resource $\ell_j$
$\eta_{i,j}$	number of times job $J_i$ may request resource $\ell_j$
$\eta_i$	total number of resource requests issued by job $J_i$

Table 2.2: Summary parameters for describing shared resources.

Locking protocols arbitrate resource requests for exclusive access issued by jobs. If a job  $J_i$  issues a request  $R_{i,j,k}$  for resource  $\ell_j$  that is unavailable, then  $J_i$  is *blocked* until  $R_{i,j,k}$  is satisfied. As depicted in Figure 2.5, job  $J_i$  is blocked during the interval  $[t_{i,j,k}^{req}, t_{i,j,k}^{sat})$ . In general, the blocked job  $J_i$  can wait for  $R_{i,j,k}$  to be satisfied by either *spinning* or *suspending*. Under spinning, job  $J_i$  remains scheduled on a processor and executes a tight polling loop (busy-waits) until  $R_{i,j,k}$  is satisfied. Under suspension,  $J_i$  relinquishes its processor and enters a suspended state, and  $J_i$  becomes ready the instant  $R_{i,j,k}$  is satisfied. The locking protocol determines whether spin- or suspension-based waiting methods are used, as well as the order in which multiple outstanding requests are satisfied. Spin-based locking protocols are commonly used when resource access times are very short, since the runtime overhead of suspending a job can exceed the time spent spinning. However, because GPUs have relatively long access times, we concern ourselves only with suspension-based locking protocols. This allows other useful work to be done while jobs wait for GPU access.

### 2.1.6.1 Priority Inversions and Progress Mechanisms

A *priority inversion* occurs whenever lower-priority work is scheduled instead of ready higher-priority work. Some sources of priority inversions are forced upon us by real-world constraints. For instance, device interrupts can be a source of priority inversions—we examine this at length later in Section 2.2.3. Resource sharing can also lead to such inversions.

**Example 2.3.** Figure 2.6(a) depicts a classic priority-inversion scenario. Here, three jobs are scheduled on a uniprocessor system. A low-priority job  $J_L$  is released at time 0. At time 5, job  $J_L$  obtains a lock on a shared resource. At time 8, a high-priority job  $J_H$  is released, preempting  $J_L$ . Job  $J_H$  requires the resource held by job  $J_L$  at time 15, so  $J_H$  blocks and is suspended from the processor; job  $J_L$  resumes execution. At time 18, a medium-priority job  $J_M$  is released and preempts job  $J_L$  because  $J_M$  has a higher priority. Job  $J_M$  continues to

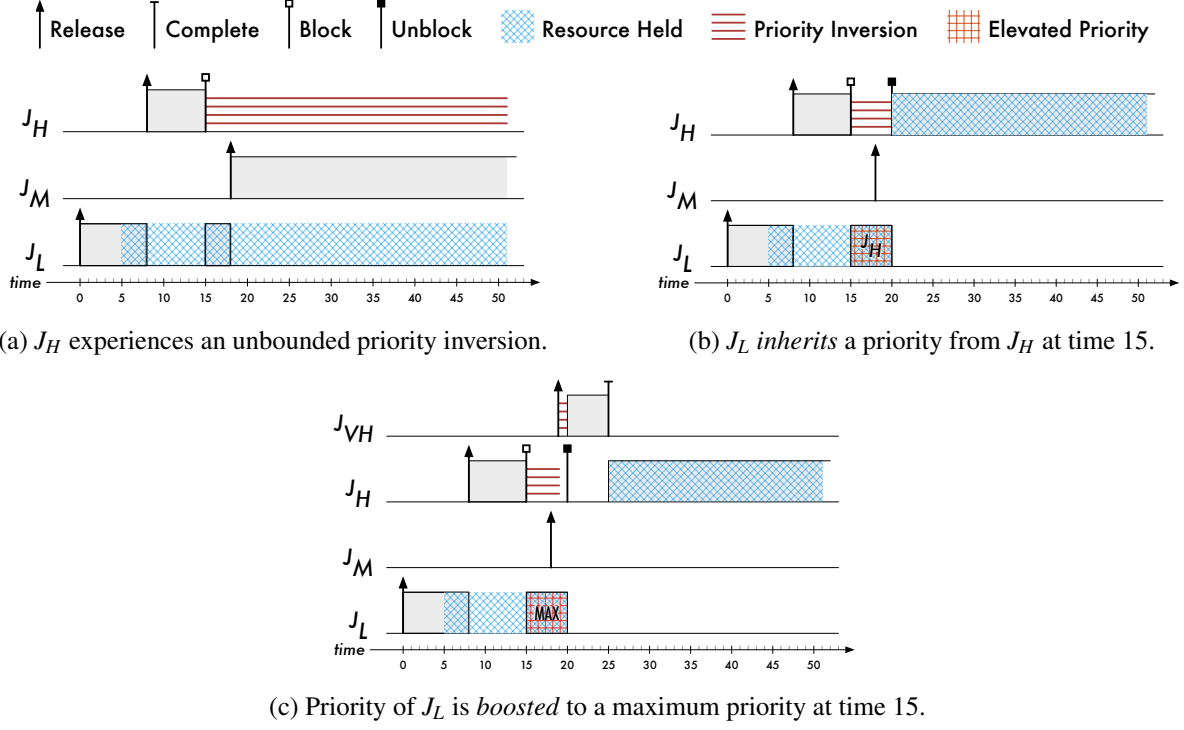


Figure 2.6: Resource sharing can lead to priority inversions. Progress mechanisms, such as priority inheritance (inset (b)) and priority boosting (inset (c)), can shorten priority inversion duration.

execute beyond the depicted schedule. Job  $J_H$  is ready to be scheduled at time 15, but it cannot since it must wait to obtain the resource held by job  $J_L$ . Thus, job  $J_H$  suffers from a priority inversion, starting at time 15.

◇

We say that the priority inversion suffered by job  $J_H$  in Figure 2.6(a) is *unbounded* because the length of the inversion depends upon the execution time of job  $J_M$ , which may be arbitrarily large. Priority inversion durations must be bounded. We accomplish this by using “progress mechanisms” that expedite the scheduling of lower-priority resource-holding jobs. That is, we intentionally *increase* the scheduling priority of a resource-holder from a *base priority* (i.e., its default priority) to a higher *effective priority*. Naturally, the resultant priority inversion bounds are a function of resource critical section lengths, as opposed to arbitrary processor demand (as was the case with job  $J_M$  in Figure 2.6(a)).

We discuss three general methods employed by real-time locking protocols to bound the duration of priority inversions due to resource sharing: priority inheritance, priority boosting, and priority donation.

**Priority Inheritance.** Under *priority inheritance* (Rajkumar, 1991; Sha *et al.*, 1990), the effective priority of a job  $J_i$  holding resource  $\ell_k$  is set to the maximum of  $J_i$ ’s base priority and the effective priority of all jobs

blocked (or that may block, depending upon the locking protocol) on  $\ell_k$ . That is, job  $J_i$  inherits the priority of the highest-priority job that is waiting for  $\ell_k$ . Job  $J_i$ 's effective priority remains elevated until  $\ell_k$  is released.

**Example 2.4.** Figure 2.6(b) depicts an example of priority inheritance. As before, job  $J_H$  requires the resource held by job  $J_L$  at time 15, so  $J_H$  blocks and is suspended from the processor; job  $J_L$  resumes execution. However, job  $J_L$  inherits the priority of job  $J_H$ , so  $J_L$  is scheduled with an effective priority of  $J_H$ . When job  $J_M$  is released at time 18, it lacks the sufficient priority to preempt  $J_L$ . Job  $J_L$  remains scheduled. Job  $J_L$  relinquishes the share resource to job  $J_H$  at time 20. With the needed resource obtained, job  $J_H$  is immediately scheduled.  $\diamond$

Priority inheritance is often viewed as the temporary transference of the priority from a high priority job to a low priority job. However, it is better to conceive of priority inheritance as a transference, occurring in the opposite direction, of *work*. That is, we may view priority inheritance as the transference of low priority work, *i.e.*, a critical section of a low priority job, to a higher-priority job. Conceptually, we may think of a job as obtaining any requested resource *immediately*—the job is never blocked. However, such a job may be required to *notionally* execute the critical sections of lower priority jobs on their behalf.<sup>6</sup> This view is embodied by a strengthened form of priority inheritance called *bandwidth inheritance* (BWI) (Lamastra *et al.*, 2001; Nogueira and Pinho, 2008). Under BWI, a resource holding job that inherits a priority also inherits the execution time *budget* of the job associated with the inherited priority. Thus, the execution time of the critical section is charged against the budget of the blocked high-priority job, not the lower-priority resource holding job. Tasks can be provisioned with enough budget to cover any budget lost due to BWI. However, budgets may be exhausted if critical sections take longer to execute than expected (or provisioned). We may take additional measures to isolate the temporal effects of such a fault. If the budget of a task with an unsatisfied resource request is exhausted, then we *abort* the task's request, *refresh* the budget of the task (possibly decreasing the task's priority), and *reissue* the aborted request (Brandenburg, 2012, 2014a).

**Priority Boosting.** Under *priority boosting* (Brandenburg and Anderson, 2013; Lakshmanan *et al.*, 2009; Rajkumar, 1990, 1991; Rajkumar *et al.*, 1988), a job  $J_i$ 's effective priority is set to the highest scheduling priority when access to a shared resource is contended, or when access is granted, depending upon the locking protocol. Job  $J_i$ 's effective priority remains elevated until  $\ell_k$  is released.

---

<sup>6</sup>The number of critical sections executed depends upon the locking protocol.

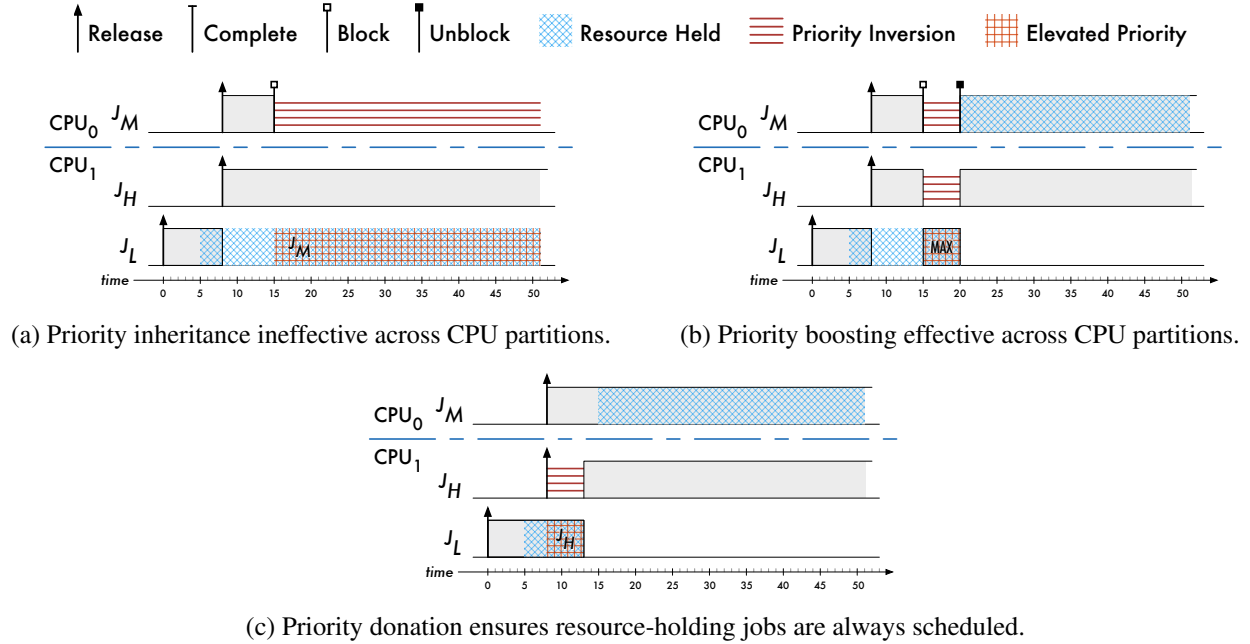


Figure 2.7: Stronger progress mechanisms may be needed in multiprocessor systems.

**Example 2.5.** Figure 2.6(c) illustrates an example of priority boosting. This figure depicts the same scenario we have studied before, except that we have added an additional very high priority job,  $J_{VH}$ , in order to illustrate an important difference between priority inheritance and priority boosting. As before, job  $J_H$  requires the resource held by job  $J_L$  at time 15, so  $J_H$  blocks and is suspended from the processor; job  $J_L$  resumes execution, but when job  $J_M$  is released at time 18, it lacks the sufficient priority to preempt  $J_L$ — $J_L$  remains scheduled. At time 19, the very high priority job  $J_{VH}$  is released. However, since job  $J_L$  has a boosted priority,  $J_{VH}$  cannot preempt  $J_L$ , either. Job  $J_L$  relinquishes the shared resource to job  $J_H$  at time 20, and job  $J_{VH}$  is immediately scheduled. Job  $J_{VH}$  completes at time 25. With the needed resource obtained, job  $J_H$  is scheduled next. ◇

Observe that in Figure 2.6(c)  $J_{VH}$  suffers from a priority inversion, even though it does not require the shared resource. This is a drawback to priority boosting: *any* job may suffer a priority inversion due to the priority boosting of another job. This can be detrimental to schedulability.

Priority boosting is implicitly used by most spin-based locking protocols, as these protocols commonly disable interrupts (disabling preemption) while spinning and within a critical section—this behavior is essential to avoiding deadlock in a spin-based locking protocol. Disabling preemption essentially gives the resource holding job a maximum priority. Priority boosting is also used in suspension-based locking

protocols. Particularly, in *multiprocessor* locking protocols where resources are shared *among* partitions or clusters of processors. In general, priority *inheritance* is an ineffective progress mechanism across partitions and clusters. This because a priority that guarantees that a job is scheduled within its own local partition or cluster (*i.e.*, the priority of a blocked job) may *not* be sufficient to guarantee that the resource holder is scheduled within a remote partition or cluster.

**Example 2.6.** The ineffectiveness of priority inheritance for resources shared across partitions or clusters is illustrated in Figure 2.7(a). Here, job  $J_M$  is partitioned to CPU<sub>0</sub>, while jobs  $J_H$  and  $J_L$  are partitioned to CPU<sub>1</sub>. Job  $J_L$  inherits a priority from  $J_M$  at time 15. However, since the priority of  $J_H$  is greater than  $J_M$ ,  $J_L$  is not scheduled.  $J_M$  experiences an unbounded priority inversion, even though it has sufficient priority to be scheduled on CPU<sub>0</sub>.

Priority boosting is effective in the same scenario, as illustrated by Figure 2.7(b). Here, the priority inversions that effect jobs  $J_H$  and  $J_M$  are bounded by the critical section length of  $J_L$ . ◇

**Priority Donation.** *Priority donation* is a recently developed progress mechanism for multiprocessor locking protocols (Brandenburg and Anderson, 2013). Priority donation is similar to priority inheritance in that a job may adopt a priority from another job. Priority donation is also similar to priority boosting in that resource holding jobs are always scheduled if they are ready. A *donor* is a job that donates its priority to a *donee* job. The effective priority of a donee job is set to that of its donor. The donation relationship between donor and donee is established upon job release of the donor.

We define two sets of pending jobs to help describe priority donation. Let  $\mathcal{J}$  denote the set of all pending jobs; these may be ready or suspended. Let the subset  $\mathcal{J}^{top-c} \in \mathcal{J}$  denote the set of pending jobs with the top- $c$  priorities (recall that  $c$  denotes the processor cluster size). Upon the release of a job  $J_d$ , if  $J_d \in \mathcal{J}^{top-c}$ , and the arrival of  $J_d$  causes a job  $J_i$  to be moved from  $\mathcal{J}^{top-c}$  to the set  $\mathcal{J} \setminus \mathcal{J}^{top-c}$ , then  $J_d$  may become a priority *donor* under the following conditions:

1. If  $J_i$  is blocked waiting for a resource or it is a resource holder, then  $J_d$  donates its priority to  $J_i$ ;  $J_i$  becomes the *donee* of  $J_d$ .
2. If  $J_i$  is a donor to a job  $J_j$ , then  $J_d$  donates its priority to  $J_j$ , ending the donor relationship between  $J_i$  and  $J_j$ , and replacing it with a relationship between  $J_d$  and  $J_j$ .

The effective priority of a donee job remains elevated until it releases any shared resources, terminating any donor relationship it may have. A priority donor may not be scheduled until its donor relationship is terminated.<sup>7</sup>

**Example 2.7.** Figure 2.7(c) illustrates an example of priority donation. Here, job  $J_L$  obtains a shared resource at time 5. The higher-priority job  $J_H$  is released at time 8. Since job  $J_L$  holds a resource, job  $J_H$  donates its priority to  $J_L$ , and  $J_L$  continues to execute. Job  $J_L$  terminates the donation relationship at time 13, when it releases the shared resource. Job  $J_H$  is immediately scheduled.  $\diamond$

In general, any job may be required to become a priority donor upon its release. As a result, any job can experience a priority inversion due to donation, since donors always suffer from a priority inversion. That is, every donor is in  $J^{top-c}$  (i.e., every donor *should* be scheduled), and no donor is ever scheduled. Priority donation and priority boosting are similar in that any job can suffer a priority inversion due to shared resources used by other jobs. However, under priority donation, this priority inversion occurs at most once per job. Whereas, under priority boosting, a job may experience *multiple* priority inversions.

Priority donation is used by the “Clustered  $k$ -exclusion  $\mathcal{O}(m)$  Locking Protocol” (CK-OMLP), developed by Brandenburg and Anderson (2013). Ward *et al.* (2012) adapted the ideas behind priority donation for the “Replica-Request Donation Global Locking Protocol” (R<sup>2</sup>DGLP). However, the R<sup>2</sup>DGLP limits donors to the set of jobs that actually share a resource. The protocol also defers the establishment of donation relationships to the moment a shared resource is requested, instead of at job release. Both the CK-OMLP and R<sup>2</sup>DGLP are a foundational element to GPUSync, so we discuss them at length in Section 2.1.7.

### 2.1.6.2 Nested Locking

A job may require exclusive access to multiple shared resources at once. This can lead to *nested* resource requests, where a task first acquires resource  $\ell_a$  and then acquires resource  $\ell_b$ . In other words, the critical section of  $\ell_b$  may be nested within the critical section of  $\ell_a$ . Arbitrary nesting of critical sections may lead to *deadlock*. The classic example of deadlock is the situation where task  $T_i$  holds resource  $\ell_a$  and blocks for access to resource  $\ell_b$ , while  $T_j$  holds resource  $\ell_b$  and blocks for access to resource  $\ell_a$ . Neither task makes progress, so the two tasks are blocked forever. Real-time correctness cannot be guaranteed for a system where

---

<sup>7</sup>Additional refinements to priority donation rules allow a donor to be scheduled under special conditions when its donee is suspended. However, this is merely a runtime optimization that does not improve schedulability analysis. We direct the interested reader to Brandenburg and Anderson (2013) for details.

deadlock is possible. There are three general approaches to supporting nested resource requests: group locks, totally-ordered nested requests, and deadlock-free locking protocols.

**Group Locks.** The first approach to manage nested locking is to define away the problem. This is done by protecting the set of nested shared resources with a single *group lock* (Block *et al.*, 2007). A task must acquire this lock if it needs to access one or more of the resources protected by the group lock. While safe, this approach limits parallelism. For example, consider the situation where three resources,  $\ell_a$ ,  $\ell_b$ , and  $\ell_c$ , are protected by a single group lock. Task  $T_i$  requires resources  $\ell_a$  and  $\ell_b$ , while task  $T_j$  only requires  $\ell_c$ . The execution of  $T_i$  and  $T_j$  is serialized when they contend for the group lock, even though they do not actually share the same resources.

**Totally-Ordered Nested Requests.** Another approach to supporting nested critical sections is to ensure that resources are acquired in an order that guarantees deadlock freedom. To do so, we enumerate all resources in a single sorted order  $\ell_1, \dots, \ell_q$ . This ordering is observed by all tasks in a system. If a task requires simultaneous access to two resources, then it must acquire  $\ell_i$  before resource  $\ell_j$ , where  $i < j$ . This generalizes to an arbitrary number of resources. It is easy to see how total ordering resolves the classic deadlock scenario. If tasks  $T_i$  and  $T_j$  both require access to resources  $\ell_a$  and  $\ell_b$ , then the tasks contend for  $\ell_a$  before they may contend for  $\ell_b$ . No task can hold  $\ell_b$  while it contends for  $\ell_a$ , so nested locking is deadlock-free. A drawback to totally ordered nested requests is that it requires disciplined programming. A given resource ordering may also be at odds with the natural flow of program code. For example, although a task  $T_i$  may require access to both resources  $\ell_a$  and  $\ell_b$ , program code may begin using  $\ell_b$  long before  $\ell_a$ . However, the total ordering requires  $\ell_a$  to be obtained early.

**Deadlock-Free Locking Protocols.** Deadlock freedom can also be guaranteed by a locking protocol algorithm. Classic (uniprocessor) real-time locking protocols that ensure deadlock freedom include the priority-ceiling protocol (PCP) (Sha *et al.*, 1990) and the stack resource policy (SRP) (Baker, 1991). These locking protocols use rules that delay access to a shared resource, even if it is available, if immediate access *may* lead to deadlock at a later time.

Another technique that can guarantee deadlock freedom is the use of *dynamic group locks* (DGLs) (Ward and Anderson, 2013). Under DGLs, a task issues requests for all resources it may require *atomically*. DGLs leverage the combined atomic request to guarantee deadlock freedom. Consider the following scenario. A task  $T_i$  requires resources  $\ell_a$  and  $\ell_b$ , while task  $T_j$  requires resources  $\ell_a$ ,  $\ell_b$ , and  $\ell_c$ .  $T_i$  issues a combined



request for  $(\ell_a, \ell_b)$ , and  $T_j$  issues a combined request for  $(\ell_a, \ell_b, \ell_c)$ . The underlying locking protocol data structures for each resource are jointly updated atomically. Under FIFO-ordered locks, resources can be granted in any order without risk of deadlock. This is because whenever tasks  $T_i$  and  $T_j$  contend for the same resources, the relative ordering between the tasks' requests is the same in every FIFO queue. Thus, access to every resource is granted in the same order. This prevents the deadlock scenario where each task waits for resources held by the other.

We must point out two important details of DGLs. First, DGLs must maintain the illusion of obtaining resources through individual requests in order to maintain sporadic task model abstractions. This means that progress mechanisms that act on behalf of a task  $T_i$  may only be active on one lock at a time. We illustrate this point with an example. Suppose task  $T_i$  waits for resources  $\ell_a$  and  $\ell_b$ , and priority inheritance is used as the progress mechanism for these locks. Either the resource holder of  $\ell_a$  or the resource holder of  $\ell_b$  may inherit the priority of  $T_i$  at any given time instant, but not both.<sup>8</sup> The second important detail of DGLs is that the underlying locking protocol *implementation* must support joint atomic updates. The data structures that manage unsatisfied lock requests are commonly protected by per-lock spinlocks that reside in the OS kernel. These spinlocks are only held while the data structures are modified. In order to support atomic DGL resource requests, all of the spinlocks related to the resources in a DGL request must be obtained before modifying the data structures of the individual locks. We have traded one multi-resource request problem (the request for DGL-protected resources) for another (the spinlocks that protect the locking protocol data structures of said resources)! We can resolve this problem in one of two ways. We may protect all locking protocol data structures with a single spinlock (*i.e.*, a group lock). This may be appropriate, since spinlocks are held for only a short duration. However, this hurts parallelism, as all concurrently issued resource requests serialize on the DGL spinlock. A better approach is to obtain the necessary spinlocks in a total order. Thankfully, this is trivial to implement in the OS kernel. Each spinlock has a unique memory address, so we obtain spinlocks in order of their memory addresses.

### 2.1.6.3 Priority-Inversion Blocking

Blocking durations must be accounted for in schedulability analysis when locking protocols are used. However, schedulability analysis must only consider blocking durations for which delays in execution cannot

---

<sup>8</sup>The only exception to this rule is when the same task holds both  $\ell_a$  and  $\ell_b$ .

be attributed to higher-priority demand (otherwise, a job without sufficient priority to be scheduled has no effect on analysis). We term this type of blocking *priority inversion blocking* (pi-blocking), and quantify the total time a job  $J_i$  may be pi-blocked with the term  $b_i$ . Brandenburg and Anderson (2013) classify analytical techniques for bounding pi-blocking due to suspension-based locking protocols as either *suspension-oblivious* (s-oblivious) or *suspension-aware* (s-aware). Under s-oblivious analysis, all suspensions, including those introduced by waiting for shared resources, are analytically treated as processor demand. Hence, a job's execution time  $e_i$  is inflated by  $b_i$  prior to performing schedulability tests such that

$$e'_i \geq e_i + b_i, \quad (2.28)$$

where  $e'_i$  denotes the safe bound on execution time used in s-oblivious schedulability analysis. Inflation essentially converts a set of dependent tasks into a set of independent tasks that can be analyzed by “normal” (locking-protocol-agnostic) schedulability tests. This approach is safe, but pessimistic in that processor time is analytically consumed by all suspensions. S-aware schedulability analysis explicitly treats  $b_i$  as suspension time. However, this treatment must be incorporated into schedulability tests. These tests are more difficult to develop, and s-aware analysis has not yet matured for all schedulers. We primarily use s-oblivious tests for global dynamic-priority JLFP schedulers. S-aware analysis is available for global fixed-priority scheduling, P-EDF, and partitioned fixed-priority scheduling (*e.g.* see Easwaran and Andersson (2009); Lakshmanan *et al.* (2009); Rajkumar (1991)).

Under global multiprocessor scheduling, if a ready job  $J_i$  is not amongst the  $m$  highest-priority ready jobs, then  $J_i$  is not scheduled, and it does not suffer from any priority inversions. This is reflected by s-aware analysis: the presence of  $m$  higher-priority ready jobs rules out the possibility of priority inversions for lower-priority jobs. In contrast, under s-oblivious analysis, the presence of  $m$  higher priority jobs, ready *or* *suspended*, rules out the possibility of priority inversions for lower-priority jobs. This fact can be exploited to implement optimal locking protocols under s-oblivious analysis, as we discuss shortly.

**Example 2.8.** Figure 2.8 illustrates the difference between s-oblivious and s-aware pi-blocking. Here, three jobs are scheduled globally across two ( $m = 2$ ) processors. Jobs  $J_H$ ,  $J_M$ , and  $J_L$  have a high, medium, and low relative priorities, respectively. Job  $J_H$  is suspended during the time interval  $[5, 15)$  while it waits for the shared resource held by job  $J_M$ . Job  $J_H$  has the highest priority, so it experiences pi-blocking under both definitions of s-oblivious and s-aware analysis. Job  $J_L$  is suspended during the time interval  $[10, 20)$ , waiting

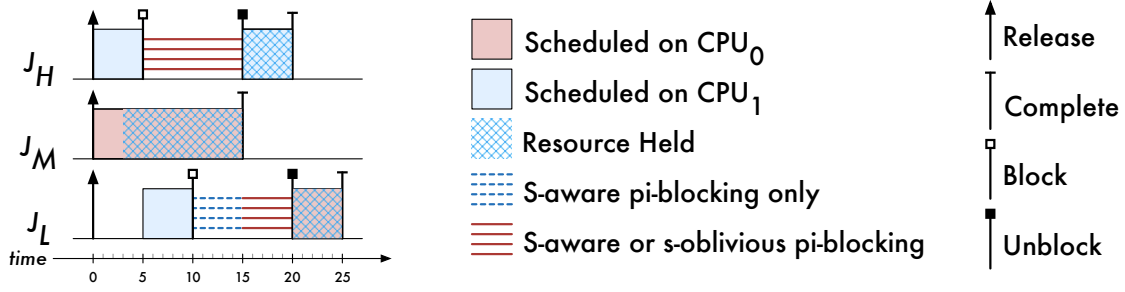


Figure 2.8: Comparison of s-oblivious and s-aware pi-blocking under global scheduling.

for the same shared resource. Because job  $J_M$  completes at time 15, job  $J_L$  experiences pi-blocking under both definitions of s-oblivious and s-aware analysis from time  $[15, 20)$  (it is among the  $m$  highest-priority pending jobs). However, job  $J_L$  experiences only s-aware pi-blocking during the time interval  $[10, 15)$  because job  $J_H$  pending but not scheduled.  $\diamond$

### 2.1.7 Multiprocessor $k$ -Exclusion Locking Protocols

Most multiprocessor locking protocols have been developed for JLFP schedulers, with most attention towards the common schedulers P-RM, P-DM, P-EDF, and G-EDF.

The *multiprocessor priority-ceiling protocol* (MPCP) (Lakshmanan *et al.*, 2009; Rajkumar, 1990) and the *distributed priority-ceiling protocol* (DPCP) (Rajkumar *et al.*, 1988) were developed for P-FP schedulers and represent the first multiprocessor real-time locking protocols. Locking protocols for P-EDF scheduling were later developed by Chen and Tripathi (1994), Gai *et al.* (2003), and Lopez *et al.* (2004). Locking protocols supporting global scheduling have developed more recently. These include the *flexible multiprocessor locking protocol* (FMLP) by Block *et al.* (2007), supporting any global JLFP scheduler, and the *parallel priority-ceiling protocol* (PPCP) by Easwaran and Andersson (2009), supporting G-FP scheduling.

Our earlier discussion on the effect of s-oblivious and s-aware analysis on pi-blocking is based upon insights of Brandenburg and Anderson (2013). These insights are a relatively new development in the analysis real-time locking protocols and allow some locking protocols to be classified as *optimal* under s-aware or s-oblivious analysis. Under s-aware analysis, a mutual exclusion locking protocol is optimal if  $b_i$  is  $\mathcal{O}(n)$ , in terms of the number of conflicting requests. In contrast, a mutual exclusion locking protocol is optimal if  $b_i$  is  $\mathcal{O}(m)$  under s-oblivious analysis; this is significant since  $m \ll n$  is common in practice. Locking protocols

from the “ $\mathcal{O}(m)$  Multiprocessor Locking Protocol” (OMLP) family of locking protocols are optimal under s-oblivious analysis (Brandenburg and Anderson, 2013; Elliott and Anderson, 2013; Ward *et al.*, 2012).

“ $K$ -exclusion” locking protocols can be used to arbitrate access to *pools* of similar or identical serially reusable resources, such as communication channels or I/O buffers.  $K$ -exclusion extends ordinary mutual exclusion (mutex) by allowing up to  $k$  tasks to simultaneously hold locks (thus, mutual exclusion is equivalent to 1-exclusion).  $K$ -exclusion has historically received little attention in the real-time community. Up until recently, only Chen (1992) had examined  $k$ -exclusion for real-time uniprocessor systems (to the best of our knowledge). However, GPU applications have renewed interest in the topic, and several real-time  $k$ -exclusion locking protocols for multiprocessor systems have resulted. These include the aforementioned CK-OMLP (Brandenburg and Anderson, 2013) and R<sup>2</sup>DGLP (Ward *et al.*, 2012), as well as a  $k$ -exclusion variant of the global FMLP-Long, called the k-FMLP (Elliott and Anderson, 2012b).<sup>9</sup>

The definition of optimality changes under  $k$ -exclusion. Under s-aware analysis, a  $k$ -exclusion locking protocol is optimal if  $b_i$  is  $\mathcal{O}(n/k)$ , in terms of the number of conflicting requests. In contrast, a  $k$ -exclusion locking protocol is optimal if  $b_i$  is  $\mathcal{O}(m/k)$  under s-oblivious analysis. The R<sup>2</sup>DGLP and CK-OMLP are optimal under s-oblivious analysis.

The k-FMLP, R<sup>2</sup>DGLP, and CK-OMLP are important to the design of GPUSync, so we discuss next the rules governing each protocol. We begin with the k-FMLP, being the simplest of the three protocols, and then discuss the R<sup>2</sup>DGLP and CK-OMLP.

### 2.1.7.1 The k-FMLP

The k-FMLP is simple extension of the global FMLP-Long to support  $k$ -exclusion.<sup>10</sup> It may be used to protect a pool of  $k$  resources shared by tasks *within* the same cluster of processors. The pi-blocking experienced by a job waiting for a resource protected by the k-FMLP is  $\mathcal{O}(n/k)$  where  $n$  is the number of tasks using the lock. The k-FMLP is designed as follows.

---

<sup>9</sup>The author of this dissertation contributed to the development of the “Optimal  $k$ -Exclusion Global Locking Protocol” (O-KGLP) (Elliott and Anderson, 2013). We do not discuss the O-KGLP since it is obsoleted by the R<sup>2</sup>DGLP, which analytically dominates the O-KGLP.

<sup>10</sup>The k-FMLP was designed by the author of this dissertation. We discuss it in this background chapter, rather than a later chapter, because the k-FMLP is a minor contribution. A detailed description and analysis of the k-FMLP may be found in the online appendix of Elliott and Anderson (2012b) at <http://www.cs.unc.edu/~anderson/papers.html>.

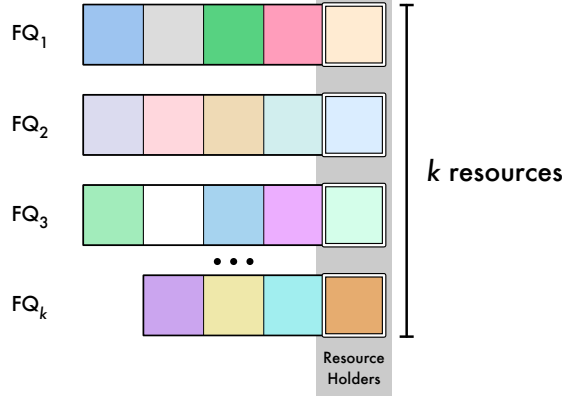


Figure 2.9: Queue structure of the k-FMLP.

**Structure.** The structure of the k-FMLP is illustrated in Figure 2.9. The k-FMLP uses  $k$  FIFO request queues, denoted  $FQ_1, \dots, FQ_k$ . Each queue is assigned to one of the  $k$  protected replicas of resource  $\ell_j$ . A job  $J_i$  enqueues a resource request  $R_{i,j,k}$  onto the queue  $FQ_x$  when the job requires a resource. A job with a request at the head of its queue is considered the holder of the associated resource and is ready to run. Jobs with blocked requests are suspended from the processor.

**Rules.** The k-FMLP may compute the length of a FIFO queue  $FQ_x$  at *runtime* with either one of two formulas. Under the simplest formulation, the length of  $FQ_x$  is given by the *number* of enqueued requests:

$$length(FQ_x) \triangleq |FQ_x|. \quad (2.29)$$

Alternatively, length may be expressed by the critical section lengths of enqueued requests:

$$length(FQ_x) \triangleq \sum_{R_i \in FQ_x} L_i, \quad (2.30)$$

where we reindex the requests in  $FQ_x$  with  $i$ . We call the formulation of Equation (2.29) *critical-section-oblivious*, and the formulation of Equation (2.30) *critical-section-aware*. We may use either formulation in the following rules that govern the k-FMLP. Let  $J_i$  denote a job that issues a request  $R_{i,j,k}$  for resource  $\ell_j$ .

- F1** When  $J_i$  issues  $R_{i,j,k}$ ,  $R_{i,j,k}$  is appended to the queue with the minimum length,  $\min_{1 \leq x \leq k} \{length(FQ_x)\}$ .  
 $J_i$  acquires the  $x^{th}$  resource when  $R_{i,j,k}$  is at the head of  $FQ_x$ .

**F2** All jobs with queued requests are suspended except for resource holders, which are ready. The effective priority of the resource holder in  $FQ_x$  is set to the maximum priority of all jobs with requests queued in  $FQ_x$ .

**F3** When  $J_i$  releases replica  $x$  of resource  $\ell_j$ ,  $R_{i,j,k}$  is dequeued from  $FQ_x$ , and the job with the next queued request in  $FQ_x$  is granted the newly available resource. If  $FQ_x$  is empty, then an arbitrary pending request (if one exists) from another queue is “stolen” (removed from its queue) and moved to  $FQ_x$ , and the stolen request is granted replica  $x$ .<sup>11</sup>

**Blocking Analysis.** We provide a summary of the s-oblivious blocking analysis presented by Elliott and Anderson (2012b). We direct the reader to that paper for the rational behind the following claims. For simplicity of presentation, we assume that each job issues one request.

By Rule F1, each request is enqueued on the shortest queue when it is issued, according to the function  $length(FQ_x)$ . Thus, the k-FMLP load-balances requests among the  $k$  resources. We denote the bound on pi-blocking that a request of job  $J_i$  for a replica of a resource  $\ell_j$  may experience under the k-FMLP with the term  $b_{i,j}^{k\text{-FMLP}}$ . Under the critical-section-oblivious formulation (Equation 2.29), request  $R_{i,j,k}$  is blocked by at most

$$b_{i,j}^{k\text{-FMLP}} = \left\lfloor \frac{n-1}{k} \right\rfloor \cdot L_j^{\max} \quad (2.31)$$

time units. No request is blocked by more than  $\left\lfloor \frac{n-1}{k} \right\rfloor$  requests. Hence, blocking under the k-FMLP is  $\mathcal{O}(n/k)$ .

Under the critical-section-aware formulation (Equation 2.30), request  $R_{i,j,k}$  is blocked by at most

$$b_{i,j}^{k\text{-FMLP}} = \frac{\sum_{T_l \in \mathcal{T} \setminus \{T_i\}} L_{l,j}^{\max}}{k} \quad (2.32)$$

time units.<sup>12</sup> Although request  $R_{i,j,k}$  may be blocked by more than  $\left\lfloor \frac{n-1}{k} \right\rfloor$  individual requests under this formulation,  $\frac{\sum_{T_l \in \mathcal{T} \setminus \{T_i\}} L_{l,j}^{\max}}{k} < \left\lfloor \frac{n-1}{k} \right\rfloor \cdot L_j^{\max}$  often holds true, so the critical-section-aware method *may* provide a tighter bound on blocking. However, the critical-section-aware method requires a more complex implementation of the k-FMLP, as it must be cognizant of the critical section lengths of enqueued requests.

<sup>11</sup>Request “stealing” does not affect worst-case blocking analysis, but it ensures efficient resource utilization at runtime.

<sup>12</sup>Tighter blocking bounds under the critical-section-aware method can be obtained by using an integer linear program to determine the *longest* the *shortest* queue may be when  $R_{i,j,k}$  is issued.

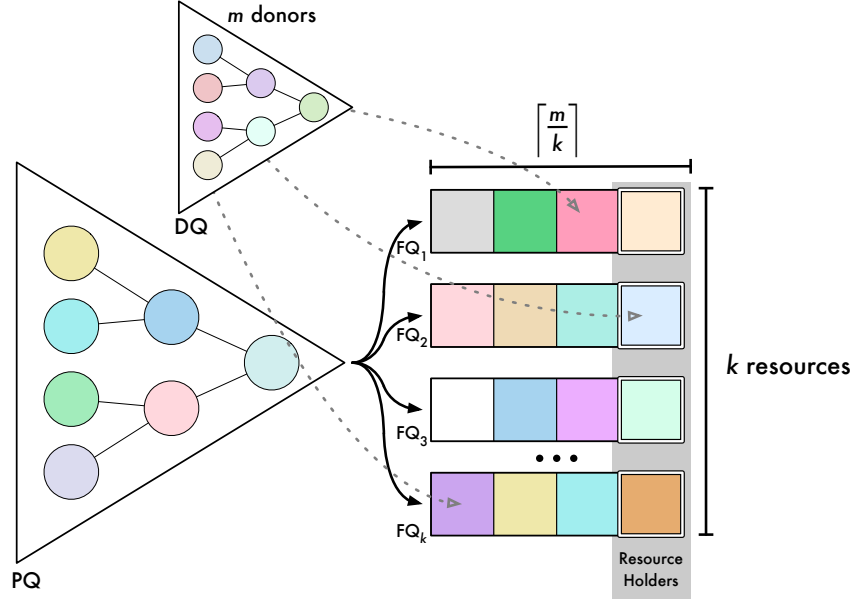


Figure 2.10: Queue structure of the  $R^2DGLP$ .

### 2.1.7.2 The $R^2DGLP$

The  $R^2DGLP$  is a  $k$ -exclusion locking protocol that is optimal under  $s$ -oblivious analysis. It may be used to protect a pool of  $k$  resources shared by tasks *within* the same cluster of processors. The pi-blocking experienced by a job waiting for a resource protected by the  $R^2DGLP$  is  $\mathcal{O}(m/k)$ .

The generality of the structure and rules that govern the  $R^2DGLP$ , as described by Ward *et al.* (2012), lends a considerable degree of leeway to the developer in the protocol’s implementation. Here, we describe a slightly simplified version to make the  $R^2DGLP$  more concrete for the reader. These simplifications do not violate the rules prescribed by Ward *et al.* The locking protocol is designed as follows.

**Structure.** The structure of the  $R^2DGLP$  is illustrated in Figure 2.10. Similar to the  $k$ -FMLP, the  $R^2DGLP$  uses  $k$  FIFO requests queues, denoted  $FQ_1, \dots, FQ_k$ . Each queue is assigned to one of the  $k$  protected replicas of resource  $\ell_j$ . A job with a request at the head of its queue is considered the holder of the associated resource and is ready to run. Jobs with blocked requests are suspended from the processor. However, unlike the  $k$ -FMLP, no FIFO queue may hold more than  $\lceil m/k \rceil$  requests. Additional requests may “overflow” into one of two priority queues, denoted PQ and DQ. (We depict PQ and DQ in Figure 2.10 with triangles, as they are efficiently implemented by heap data structures.) Requests with a sufficiently high priority are inserted into DQ, while others are inserted into PQ. PQ is organized by priority-order. DQ is organized in

*reverse-priority-order* (*i.e.*, the lowest-priority request appears at the head of the queue). As we describe shortly, DQ may hold at most  $m$  requests.

**Rules.** Let  $J_i$  denote a job that issues a request  $R_{i,j,k}$  for resource  $\ell_j$ . The priority of  $R_{i,j,k}$  is equivalent to the base priority of  $J_i$ . We denote the set of incomplete requests with the  $m$ -highest priorities by  $\mathcal{M}$ . The R<sup>2</sup>DGLP makes use of the relaxed notion of priority donation as discussed at the end of Section 2.1.6.1, whereby the establishment of a donation relationship is deferred to the moment a shared resource is requested, rather than at job release. We denote the set of requests that have a priority donor (*i.e.*, the donees) by  $\mathcal{d}$ . The length of  $\text{FQ}_x$  is the number of enqueued requests in  $\text{FQ}_x$ , given by the function

$$\text{length}(\text{FQ}_x) \triangleq |\text{FQ}_x|. \quad (2.33)$$

With these definitions, the R<sup>2</sup>DGLP operates under the following rules.

**R1** When  $J_i$  issues  $R_{i,j,k}$ :

- (a)  $R_{i,j,k}$  is enqueued on the shortest FIFO queue,  $\text{FQ}_x$ , if  $\text{length}(\text{FQ}_x) \leq \lceil m/k \rceil$ .
- (b) else if  $R_{i,j,k} \notin \mathcal{M}$ , then  $R_{i,j,k}$  is enqueued in PQ.
- (c) else  $R_{i,j,k}$  is enqueued in DQ. If the number of requests in DQ is no greater than  $m$  after  $R_{i,j,k}$  is enqueued, then the priority of  $R_{i,j,k}$  is donated to an arbitrary request in the set  $\bigcup_{x=1}^k \{R | R \in \text{FQ}_x\} \setminus \mathcal{d} \setminus \mathcal{M}$  (*i.e.*, any request in an FQ without a donor and is not among the requests in  $\mathcal{M}$ ). Otherwise, the lowest-priority request  $R^L$  in DQ (which cannot be  $R_{i,j,k}$  by Rule R1b), is moved to PQ, and  $R_{i,j,k}$  becomes the priority donor to the donee of  $R^L$ . (Donor relationships are depicted in Figure 2.10 by dashed arrows.)

**R2**  $R_{i,j,k}$  is satisfied when it is at the head of  $\text{FQ}_x$ .

**R3**  $J_i$  suspends until request  $R_{i,j,k}$  is satisfied.

**R4** The job with a request at the head of  $\text{FQ}_x$  inherits the highest effective priority (which could be a donated priority) of any request in  $\text{FQ}_x$ .

**R5**  $R_{i,j,k}$  is dequeued from  $\text{FQ}_x$  when  $J_i$  releases the replica  $x$ . If  $R_{i,j,k}$  has a priority donor  $R^D$ , then  $R^D$  is removed from DQ (by Rule R1c all donors must be in DQ) and enqueued on  $\text{FQ}_x$ . The job that issued



$R^D$  no longer donates its priority to the job of  $R_{i,j,k}$ . Otherwise, the request at the head of PQ (if it exists) is removed from PQ and enqueued on  $FQ_x$ .

**Blocking Analysis.** We provide a summary of the blocking analysis presented by Ward *et al.* (2012). We direct the reader to that paper for the rational behind the following claims. For simplicity of presentation, we assume that each job issues one request.

We begin by bounding the *number* of other requests that may pi-block a request  $R_{i,j,k}$  within each queue under s-oblivious analysis.

*Pi-blocking in  $FQ_x$ :* Rule R1a ensures that the maximum length of  $FQ_x$  is  $\lceil m/k \rceil$ . No request in  $FQ_x$  is pi-blocked by more than  $\lceil m/k \rceil - 1$  other requests.

*Pi-blocking in  $DQ$ :* Rule R5 ensures that a donor request in  $DQ$  is moved to  $FQ_x$  once their donee request completes. A request in  $DQ$  is pi-blocked by no more than  $\lceil m/k \rceil$  other requests.

*Pi-blocking in  $PQ$ :* Rules R1b and R1c ensure that  $R_{i,j,k} \notin \mathcal{M}$  holds at the time  $R_{i,j,k}$  enters PQ.  $R_{i,j,k}$  may only enter  $\mathcal{M}$  when a resource holder releases a replica, *i.e.*, when the request of a resource holder,  $R_x$ , exits from  $FQ_x$ .  $R_x \in \mathcal{M}$  must hold if the dequeue of  $R_x$  from  $FQ_x$  promotes  $R_{i,j,k}$  into  $\mathcal{M}$ . By Rule R1c,  $R_x$  cannot have a priority donor. Thus, by Rule R5, a request from PQ is moved into  $FQ_x$ .  $R_{i,j,k}$  must have the highest priority among all requests in PQ since  $R_{i,j,k} \in \mathcal{M}$  while  $\{R \mid R \in PQ \wedge R \neq R_{i,j,k}\} \cap \mathcal{M} = \emptyset$ . By Rule R5,  $R_{i,j,k}$  is removed from PQ and enqueued on  $FQ_x$ . No request in PQ experiences pi-blocking under s-oblivious analysis, because the moment such a request *could* experience pi-blocking, it is moved to an FQ.

Summing the pi-blocking that can be incurred by  $R_{i,j,k}$  as it moves through the queues, the maximum number of other requests that may pi-block a request is

$$2 \cdot \left\lceil \frac{m}{k} \right\rceil - 1. \quad (2.34)$$

We denote the bound on pi-blocking that a request of job  $J_i$  for a replica of a resource  $\ell_j$  may experience under the  $R^2DGLP$  with the term  $b_{i,j}^{R^2DGLP}$ . By Equation (2.34),

$$b_{i,j}^{R^2DGLP} = \left( 2 \cdot \left\lceil \frac{m}{k} \right\rceil - 1 \right) \cdot L_j^{max}. \quad (2.35)$$

Hence, pi-blocking under the R<sup>2</sup>DGLP is  $\mathcal{O}(m/k)$ —optimal under s-oblivious analysis. Equation (2.35) provides a coarse-grain bound on pi-blocking. We examine derivation for finer-grained blocking bounds in Chapter 4.

### 2.1.7.3 The CK-OMLP

The CK-OMLP, like the R<sup>2</sup>DGLP, is a  $k$ -exclusion locking protocol that is optimal under s-oblivious analysis. Unlike the R<sup>2</sup>DGLP, the CK-OMLP also supports the protection of a pool of  $k$  resources shared by tasks *across* processor clusters. The pi-blocking experienced by *any* job in a cluster where resources protected by the CK-OMLP is  $\mathcal{O}(m/k)$ . The CK-OMLP is designed as follows.

**Structure.** The structure of the CK-OMLP is illustrated in Figure 2.11. The CK-OMLP uses a single FIFO queue, denoted FQ, that holds a maximum of  $m - k$  unsatisfied resource requests. The requests of the  $k$  resource holders are *not* kept in FQ. The CK-OMLP relies upon additional *scheduler* data structures that track the  $c$ -highest priority incomplete jobs within each processor cluster. Min-heaps are an efficient data structure for such bookkeeping, so these scheduler data structures are depicted by triangles in Figure 2.11. We denote the set of the  $c$ -highest priority incomplete jobs with the  $a^{th}$  cluster by  $\mathcal{C}^a$ . A job may be in  $\mathcal{C}^a$  while also waiting for, or holding, a resource replica.

**Rules.** The CK-OMLP operates under the following rules. Let  $J_i$  denote a job in the  $a^{th}$  cluster that issues a request  $R_{i,j,k}$  for resource  $\ell_j$ .

- C1**  $J_i$  receives a donated priority from a donor job in  $\mathcal{C}^a$  if  $J_i \notin \mathcal{C}^a$ , pursuant to the description of priority donation in Section 2.1.6.1, while  $R_{i,j,k}$  is incomplete.
- C2**  $J_i$  acquires the replica  $x$  when  $J_i$  issues  $R_{i,j,k}$  if such a replica is available. Otherwise,  $R_{i,j,k}$  is enqueued on FQ and  $J_i$  suspends.
- C3** When  $J_i$  releases the replica  $x$ , the pending request at the head of FQ (if it exists) is dequeued, and the associated job acquires  $x$ .

**Blocking Analysis.** We provide a summary of the blocking analysis presented by Brandenburg and Anderson (2013). We direct the reader to that paper for the rational behind the following claims. For simplicity of presentation, we assume that each job issues at most one request per resource.

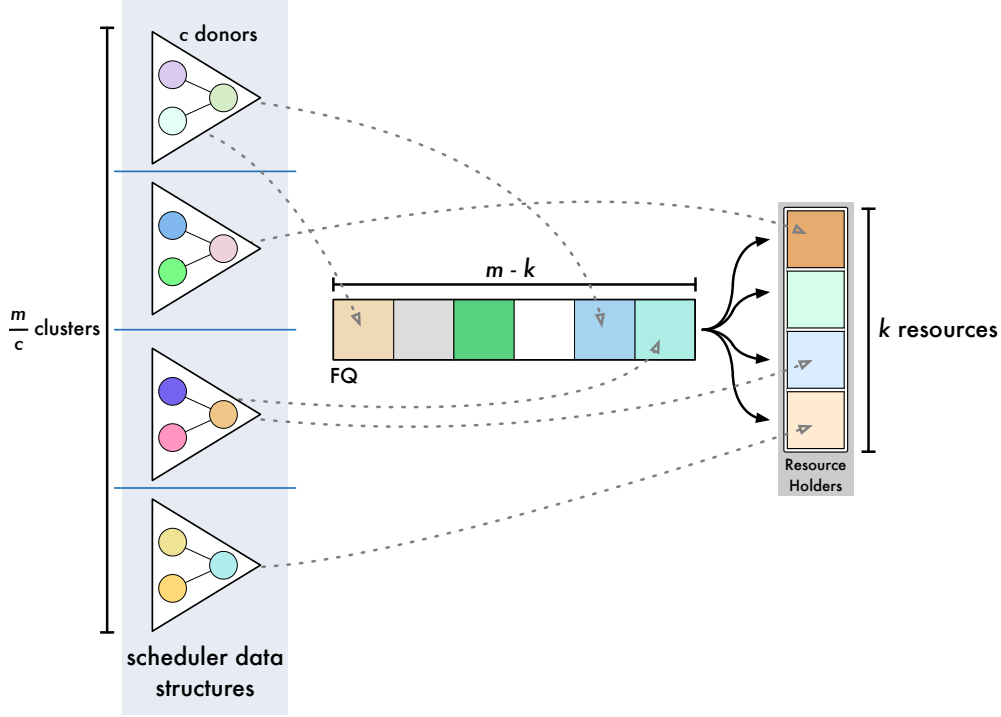


Figure 2.11: Queue structure of the CK-OMLP.

Under the CK-OMLP, a job may experience both *direct* and *indirect* pi-blocking. A job may experience direct pi-blocking while it is blocked for a shared resource. A job may experience indirect blocking while it acts as a priority donor. We consider bounds on direct and indirect blocking, in turn.

The priority donation rule, Rule C1, ensures that there are at most  $m$  incomplete requests. Since the  $k$  resource holders are not kept in FQ, FQ has a maximum length of  $m - k$ . We denote the bound on direct pi-blocking with the term  $b_{i,j}^{\text{CK-OMLP-D}}$ . By Rule C1, resource holders are always scheduled, so requests are satisfied at a rate of at least  $k$  requests per  $L_j^{\max}$ -units-of-time. Thus, by this property and Rule C3, direct pi-blocking is bounded by

$$b_{i,j}^{\text{CK-OMLP-D}} = \left\lceil \frac{m-k}{k} \right\rceil \cdot L_j^{\max} = \left( \left\lceil \frac{m}{k} \right\rceil - 1 \right) \cdot L_j^{\max}. \quad (2.36)$$

A job experiences indirect pi-blocking while it acts as a priority donor. We denote the bound on indirect pi-blocking due to resource  $\ell_j$  with the term  $b_{i,j}^{\text{CK-OMLP-I}}$ . This duration is bounded by the maximum time that a request may be waiting in FQ, plus the critical section length of that request. Thus, indirect pi-blocking due

to resource  $\ell_j$  is bounded by

$$b_{i,j}^{\text{CK-OMLP-}I} = b_{i,j}^{\text{CK-OMLP-}D} + 1 \cdot L_j^{\max} = \left\lceil \frac{m}{k} \right\rceil \cdot L_j^{\max}. \quad (2.37)$$

There may exist multiple pools of resources that are each protected by a different instance of the CK-OMLP, so  $b_{i,j}^{\text{CK-OMLP-}I}$  does not bound indirect pi-blocking due to *all* such resources. We denote the set of resources accessed by tasks within the  $a^{\text{th}}$  cluster by  $\ell^a$ . We denote the bound on *total* indirect pi-blocking by  $b_i^{\text{CK-OMLP-}I}$ , which is

$$b_i^{\text{CK-OMLP-}I} = \max_{\ell_j \in \ell^a} \left\{ b_{i,j}^{\text{CK-OMLP-}I} \right\}. \quad (2.38)$$

We denote the bound on the total pi-blocking experienced by a job under the CK-OMLP with the term  $b_i^{\text{CK-OMLP}}$ . For jobs that do *not* issue requests,

$$b_i^{\text{CK-OMLP}} = b_i^{\text{CK-OMLP-}I}. \quad (2.39)$$

Let  $\ell_i^a$  denote the subset of resources accessed by job  $J_i$  that is scheduled within the  $a^{\text{th}}$  cluster. For jobs that do issue requests resource requests under the CK-OMLP,

$$b_i^{\text{CK-OMLP}} = b_i^{\text{CK-OMLP-}I} + \sum_{\ell_j \in \ell_i^a} b_{i,j}^{\text{CK-OMLP-}D}. \quad (2.40)$$

This concludes our review of real-time  $k$ -exclusion locking protocols.

### 2.1.8 Accounting for Overheads in Schedulability Tests

The schedulability tests we discussed in Section 2.1.5 assume that all scheduling decisions and actions are instantaneous. However, this is impossible to achieve in the real world. Simply, scheduling algorithms take time to execute. Moreover, scheduling decisions have side-effects. For instance, a job's execution time regularly increases with every preemption due to the loss of cache affinity. Basic OS functions, such as processor and device interrupt handling, introduce additional delays to real-time jobs. Contention for shared hardware, such as a system memory bus or caches, cause concurrently executing jobs to interfere with one another. Collectively, we refer to these costs, as well as others, as *system overheads*. *Overhead-aware* schedulability tests are those that incorporate system overheads into analysis.

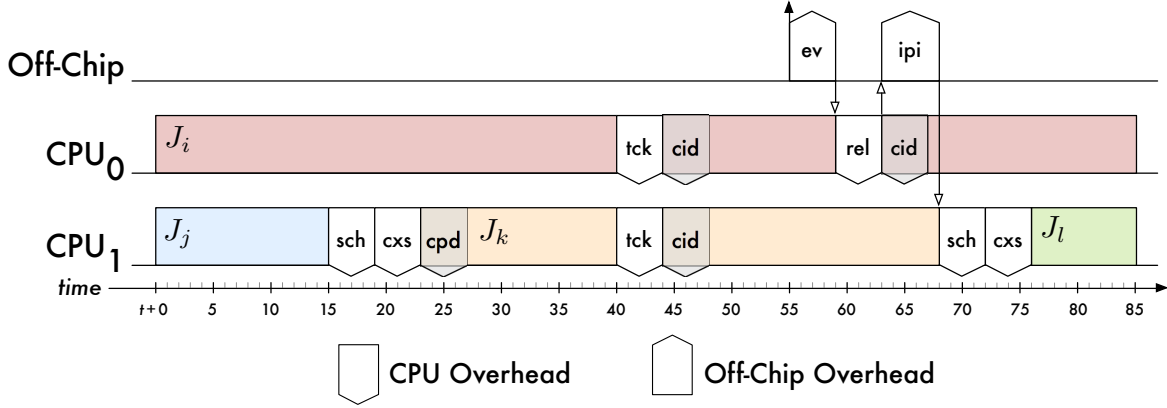


Figure 2.12: Schedule depicting system overheads.

We now describe the “preemption-centric” method of Brandenburg (2011b), which we use for overhead-aware schedulability analysis in this dissertation. We first focus our attention on the case where tasks are independent, *i.e.*, they share no resources besides processors. We discuss additional methods for account for overheads due to resource sharing, thereafter.

### 2.1.8.1 Preemption-Centric Accounting

Figure 2.12 depicts a schedule for four jobs on a system with two CPUs. In this illustrative example, we do not need to consider task parameters or the particular scheduling algorithm in use to study the relevant overheads. Prior to time  $t$ , assume the following: jobs  $J_i$  and  $J_j$  were scheduled on  $\text{CPU}_0$  and  $\text{CPU}_1$ , respectively; job  $J_k$  had been scheduled, but was either preempted or suspended from execution; and a hardware timer has been set to time 55 to trigger the release of job  $J_l$ . We observe several different overheads in Figure 2.12:

- *Scheduling* (*sch*) overheads are incurred when a job completes (time 15) or when a newly arrived job must preempt a scheduled job (time 68).
- A *release* (*rel*) overhead is incurred when a job becomes ready for execution (time 59). This may occur when a new job is released, or when an incomplete job resumes after having voluntarily relinquished a CPU.
- A scheduler *tick* (*tck*) overhead is incurred at regular intervals (time 40). In Figure 2.12, we assume that the scheduler ticks of the CPUs are synchronized, but this is not strictly necessary. Within the tick

handler, the OS may reevaluate scheduler state or perform bookkeeping functions, such as tracking the execution time of currently scheduled jobs.

- *Context switch (cxs)* overhead is incurred whenever the scheduler switches from one job to another (times 19 and 72). This process takes time as the scheduler must save program state (*e.g.*, CPU register values) of the old job, restore the state of the new job.
- A *cache preemption/migration delay (cpd or CPMD)* overhead is incurred when a previously scheduled job resumes from execution (time 23). This overhead reflects the fact that such a job may have to reload data that had been previously cached. In practice, this overhead is incurred *incrementally* as a job executes. However, without exact knowledge of the state of the resuming job or caches, we analytically assume a single worst-case overhead.
- A *cache interrupt delay (cid)* overhead is incurred after every interrupt is handled by a CPU (times 44 and 63). The interrupt handler should be relatively light-weight in terms of execution cost, leaving the cached state of the interrupted job relatively intact, so we account for cache interrupt delays separately from cache preemption/migration delays.
- An *event (ev)* overhead accounts for the latency between the occurrence of an event, *e.g.*, the firing of a hardware timer (time 55), and when that event is communicated to a CPU (time 59). Event overheads do not consume CPU time, so Figure 2.12 depicts this overhead as occurring in parallel “off-chip.”
- *Interprocessor interrupt (ipi)* overheads are incurred in cooperative multiprocessor schedulers where a CPU may make scheduling decisions for other CPUs. A CPU communicates such scheduling decisions to the target CPU by sending an interprocessor interrupt (IPI) to a target CPU. The IPI overhead captures the latency between when an IPI is sent (time 63) and when it is received (time 68). Like the event overhead, this overhead does not consume CPU time, so Figure 2.12 depicts this overhead as occurring in parallel “off-chip.”

The overhead durations reflected in Figure 2.12 are merely illustrative. The actual costs of each overhead vary and depend upon different factors. For instance, context switch overheads depend primarily upon the capabilities of the underlying processor and memory subsystem. Scheduling overheads, on the other hand, depend upon the runtime complexity of the scheduling algorithm and its implementation. Cache-related

Parameter	Meaning
$\Delta^{sch}$	scheduler overhead
$\Delta^{rel}$	release overhead
$\Delta^{tck}$	scheduler tick overhead
$\Delta^{cxs}$	context switch overhead
$\Delta^{cpd}$	cache preemption/migration delay
$\Delta^{cid}$	cache interrupt delay
$\Delta^{ev}$	event latency
$\Delta^{ipi}$	interprocessor interrupt latency
$Q$	scheduler tick quantum
$T_0^{tck}$	virtual task modeling the scheduler tick interrupt
$T_i^{irq}$	virtual task modeling the release interrupt of task $T_i$
$c^{pre}$	cost of one preemption due to periodic interrupts

Table 2.3: Summary of parameters used in preemption-centric overhead accounting.

overheads depend upon the memory footprint and data access patterns of the jobs themselves. Each overhead may be quantified though empirical measurement. We explore this methodology further in Chapter 4.

We now present the formalisms developed by Brandenburg (2011b) to integrate the above overheads into schedulability analysis. The general strategy of preemption-centric overhead accounting is to *inflate* a task’s WCET with a charge that accounts for the overheads that the task may experience in a worst-case scenario. This provides a safe upper-bound on what a task may experience at runtime. We denote each overhead with the symbol  $\Delta$ , with the type of overhead expressed by the label in superscript. Table 2.3 contains a summary of the parameters we use in overhead accounting.

*Periodic* interrupt sources are modeled as virtual tasks that always preempt non-virtual tasks.<sup>13</sup> The scheduler tick interrupt is modeled by the virtual task  $T_0^{tck}$ . Release interrupts are modeled by a per-task virtual task  $T_i^{irq}$ . These tasks take the following values:

$$\begin{array}{lll}
e_0^{tck} \triangleq \Delta^{tck} + \Delta^{cid} & p_0^{tck} \triangleq Q & u_0^{tck} \triangleq \frac{\Delta^{tck} + \Delta^{cid}}{Q} \\
e_i^{irq} \triangleq \Delta^{rel} + \Delta^{cid} & p_i^{irq} \triangleq p_i & u_i^{irq} \triangleq \frac{\Delta^{rel} + \Delta^{cid}}{p_i}
\end{array}$$

Here,  $Q$  denotes the period of the scheduler tick, or *quantum*. The value of  $Q$  is set by the OS. Also, observe that  $T_i^{irq}$  is periodic, even if its associated task  $T_i$  is sporadic. This is done to safely bound utilization loss due to release interrupts.

---

<sup>13</sup>We emphasize that this virtual-task approach can only be applied to interrupts that may be modeled with a periodic arrival pattern. Interrupt overheads due to non-periodic interrupt sources, such as GPUs, must be accounted for in a different manner.

The total cost of one preemption due to these periodic interrupts is given by:

$$c^{pre} \triangleq \frac{e_0^{tck} + \Delta^{ev} \cdot u_0^{tck} + \sum_{1 \leq i \leq n} (\Delta^{ev} \cdot u_i^{irq} + e_i^{irq})}{1 - u_0^{tck} - \sum_{1 \leq i \leq n} u_i^{irq}}. \quad (2.41)$$

We transform the task set  $\mathcal{T}$  into the task set  $\mathcal{T}'$  by inflating task execution times and shrinking task periods, such that:

$$e'_i \geq \frac{e_i + 2(\Delta^{sch} + \Delta^{cxs}) + \Delta^{cpd}}{1 - u_0^{tck} - \sum_{1 \leq i \leq n} u_i^{irq}} + 2c^{pre} + \Delta^{ipi}, \quad (2.42)$$

$$p'_i \leq p_i - \Delta^{ev}, \quad (2.43)$$

$$d'_i \leq d_i - \Delta^{ev}. \quad (2.44)$$

Once we have obtained  $\mathcal{T}'$  using the above formulations by Brandenburg (2011b), we perform schedulability analysis upon the task set  $\mathcal{T}'$ , instead of  $\mathcal{T}$ , to safely test for schedulability.

### 2.1.8.2 Locking Protocol Overheads

Runtime overheads due to locking protocols must also be accounted for in overhead-aware schedulability analysis. For instance, we must consider the time it may take to execute locking protocol logic to issue a resource request. We must also consider scheduling decisions that may result from changes in job priorities due to priority inheritance, boosting, or donation. We briefly discuss Brandenburg's preemption-centric overhead accounting methodology to account for such overheads. We direct the reader to Brandenburg (2011b) for a full explanation of the following analysis.

Suspension-based locking protocols introduce four additional types of overheads:

- *System call entry (sci)* overhead is incurred when a job invokes the OS in order to issue a resource request.
- *System call return (sco)* overhead is incurred when a resource request is satisfied, and the resource-holding job resumes execution, switching control from the OS to the job.
- *Lock (lk)* overhead represents the execution cost of issuing a request under a given locking protocol.
- *Unlock (ulk)* overhead represents the execution cost of releasing a held resource.



Parameter	Meaning
$\Delta^{sci}$	system call entry overhead
$\Delta^{sco}$	system call return overhead
$\Delta^{lk}$	lock overhead
$\Delta^{ulk}$	unlock overhead

Table 2.4: Summary of locking protocol overheads considered by preemption-centric accounting.

These overheads are summarized in Table 2.4. Overheads affect *both* the execution time of jobs and increase the lengths of critical sections.

**Overheads and WCET.** Brandenburg prescribes the following additional inflation to job WCET to account for resource requests overheads, assuming suspension-based locking protocols:

$$e'_i \geq e_i + \eta_i \cdot (2\Delta^{sci} + 2\Delta^{sco} + \Delta^{lk} + \Delta^{ulk} + 3\Delta^{sch} + 3\Delta^{cxs} + 2\Delta^{cpd} + \Delta^{ipi}) \quad (2.45)$$

We charge two instances of system call overheads ( $\Delta^{sci}$  and  $\Delta^{sco}$ ) to account for the calls made by a job to request and release a resource, respectively. Likewise, we charge lock ( $\Delta^{lk}$ ) and unlock ( $\Delta^{ulk}$ ) overheads to account for the locking protocol's handling of these calls. We charge two sets of scheduling, context switch, and cache affinity loss overheads ( $2\Delta^{sch} + 2\Delta^{cxs} + 2\Delta^{cpd}$ ) to cover the cost of self-suspension when job  $J_i$  is blocked. We incorporate the overhead of one IPI ( $\Delta^{ipi}$ ) to account for the latency of waking up  $J_i$  after its request has been satisfied. Finally, we charge an *additional* set of scheduling and context switch overheads ( $\Delta^{sch} + \Delta^{cxs}$ ) to account for a potential change in priority of the resource-holding task triggered by the blocked request of  $J_i$ . (We do not make this charge for locking protocols where resource-holding jobs cannot be preempted, such as the CK-OMLP.) We inflate  $e_i$  using Equation (2.45) prior to the application of Equation (2.42). In the application of Equation (2.42), we must also include an additional charge of  $\eta_i \cdot c^{pre}$  if processors are not shielded from interrupts, since tick and release interrupts may further delay the resumption of job  $J_i$  after it has acquired its resource.

**Overheads and Critical Section Lengths.** Overheads may also be incurred during the execution of a critical section, effectively increasing the length of the critical section. This, in turn, affects blocking analysis. We inflate critical section lengths to account for these overheads prior to blocking analysis. Brandenburg inflates

each critical section length for suspension-based locking protocols, such that:

$$L'_{i,j,k} \geq L_{i,j,k} + 2\Delta^{sch} + 2\Delta^{cxs} + \Delta^{sci} + \Delta^{sco} + \Delta^{ulk} + \Delta^{ipi}. \quad (2.46)$$

We charge an IPI overhead ( $\Delta^{ipi}$ ) to account for the delay in notifying a suspended job that it has obtained its needed resource and that it may execute (*i.e.*, the delay in waking up the waiting job). One set of scheduling and context switch ( $\Delta^{sch} + \Delta^{cxs}$ ) overheads account for the time it takes a job to resume execution after suspension. The remaining scheduling and context switch overheads account for the situation where a resource-holding job may be preempted by another resource-holding job. (As in Equation (2.45), we do not make this charge for locking protocols where resource-holding jobs cannot be preempted.) Finally, the remaining overheads account for the time taken to free a resource ( $\Delta^{sci} + \Delta^{sco} + \Delta^{ulk}$ ).

### 2.1.9 Integration of PI-Blocking and Overhead Accounting

In this section, we tie together the pi-blocking and overhead accounting methods we discussed above into a six-step procedure for performing overhead-aware schedulability tests. Our procedure is tailored to s-oblivious analysis and the suspension-based locking protocols we use in this dissertation. We refer to values that are computed in each step with a superscript. For example,  $e_i^{[j]}$  denotes the bound for job execution time of task  $T_i$  computed in the  $j^{th}$  step. We restate several formulas in order to give a consolidated presentation of the procedure.

#### Step 1: Inflate Critical Sections

We begin our procedure by inflating the critical section lengths of each resource request:

$$L_{i,j,k}^{[1]} \triangleq L_{i,j,k} + 2\Delta^{sch} + 2\Delta^{cxs} + \Delta^{sci} + \Delta^{sco} + \Delta^{ulk} + \Delta^{ipi}. \quad (2.47)$$

This step corresponds to the application of Equation (2.46).

#### Step 2: Bound S-Oblivious PI-Blocking

Using the inflated critical section lengths computed in the prior step (*i.e.*,  $L_{i,j,k}^{[1]}$ ), we compute bounds on s-oblivious pi-blocking in according to the blocking analysis techniques prescribed by the locking protocols that we employ. For example, we may bound pi-blocking under the k-FMLP using Equa-

tion (2.30). For each job, we compute the total bound on pi-blocking each job may experience, denoted by  $b_i^{[2]}$ .

### Step 3: Convert S-Oblivious PI-Blocking to Execution Time

We inflate job execution time to incorporate our bounds on pi-blocking:

$$e_i^{[3]} \triangleq e_i + b_i^{[2]} \quad (2.48)$$

### Step 4: Account for Locking Protocol Overheads

We inflate job execution time to account for the execution of locking protocol algorithms and associated scheduling costs:

$$e_i^{[4]} \triangleq e_i^{[3]} + \eta_i \cdot (2\Delta^{sci} + 2\Delta^{sco} + \Delta^{lk} + \Delta^{ulk} + 3\Delta^{sch} + 3\Delta^{cxs} + 2\Delta^{cpd} + \Delta^{ipi}) \quad (2.49)$$

This step corresponds to the application of Equation (2.45).

### Step 5: Account for General Overheads

We inflate job execution time according to the preemption-centric method:

$$e_i^{[5]} \triangleq \frac{e_i^{[4]} + 2(\Delta^{sch} + \Delta^{cxs}) + \Delta^{cpd}}{1 - u_0^{tck} - \sum_{1 \leq i \leq n} u_i^{irq}} + (2 + \eta_i) \cdot c^{pre} + \Delta^{ipi}, \quad (2.50)$$

$$p_i^{[5]} \triangleq p_i - \Delta^{ev}, \quad (2.51)$$

$$d_i^{[5]} \triangleq d_i - \Delta^{ev}. \quad (2.52)$$

This step corresponds to the application of Equations (2.42), (2.43), and (2.44). As we discussed in the description of Equation (2.45), we incorporate an additional charge of  $\eta_i \cdot c^{pre}$  because we assume processors are not shielded from interrupts.

### Step 6: Perform Schedulability Analysis

Steps 1 through 5 give rise to a transformed task set,  $\mathcal{T}^{[5]}$ , that accounts for pi-blocking and overheads. We analyze  $\mathcal{T}^{[5]}$  using “normal” (*i.e.*, overhead- and locking-protocol-agnostic) schedulability analysis. For example, we may test  $\mathcal{T}^{[5]}$  for bounded deadline tardiness under global fair-lateness (G-FL)

scheduling using Equation (2.23). If  $\mathcal{T}^{[5]}$  has bounded deadline tardiness, then  $\mathcal{T}$  is also has bounded deadline tardiness.

This concludes our discussion of real-time task models, locking protocols, and overhead-aware real-time schedulability analysis. We now shift our attention towards real-time operating systems and the services that they provide to user-level applications.

## **2.2 Real-Time Operating Systems**

In this section, we discuss the role of real-time operating systems (RTOSs) in the support of real-time applications. We begin with a brief discussion of basic RTOS requirements that must be met in order to support the realization of sound real-time systems. We then discuss LITMUS<sup>RT</sup>, an RTOS that we extend in order to meet the needs of our real-time GPGPU applications. Finally, we conclude with an in-depth discussion of interrupt handling techniques in general purpose and real-time OSs. This topic is relevant to us as GPUs use interrupts to signal the completion of operations.

### **2.2.1 Basic RTOS Requirements**

An RTOS is the underlying software that manages hardware resources and coordinates the execution of user applications. It must meet the needs of the real-time applications that rely upon it. Of course, these needs are application-specific. Due to the wide variety of real-time applications, there are a wide variety of RTOSs. RTOSs range from small microcontroller environments to fully-featured OSs that are as capable as, if not more capable than, general purpose OSs. Here, we only discuss basic RTOS requirements. We direct the interested reader to a taxonomy and thorough survey of modern RTOSs by Brandenburg (2011b) for a more in-depth discussion of RTOS capabilities.

Fundamentally, an RTOS is responsible for providing a deterministic and predictable environment for user applications, while fulfilling other operating system requirements (such as memory management and file system support). System designers require a high degree of confidence that the timing constraints of their real-time applications are met. This is only possible if the underlying RTOS ensures deterministic and predictable behavior, as far as it is able. These behaviors are predominantly realized by the RTOS scheduler, which allocates processor time to user applications and other system services. The scheduler may employ one or more real-time scheduling algorithms, such as those we discussed in Section 2.1.4.

System designers design and implement their application software in accordance to a real-time task model, such as the sporadic or rate-based task models, that is supported by the RTOS scheduler. The designer provisions tasks with resources (*e.g.*, job execution time) and specifies task execution rates (*e.g.*, task periods). With this information, the designer may use schedulability analysis to *model* the real-time behaviors of their system. Analysis provides feedback to the designer on whether their implemented system, as modeled, meets application timing requirements. We stress that schedulability analysis only *approximates* real system behavior and that nothing in analysis *forces* the implemented software to behave as modeled. Analytical results hold only if the following requirements are met:

1. The RTOS scheduler adheres to the real-time scheduling algorithm modeled by the analysis.
2. Application behaviors are predictable.

The first requirement is achieved through the thoughtful design and implementation of the RTOS. The RTOS must employ strategies that eliminate or minimize priority inversions (recall that, by definition, a priority inversion is a deviation from the real-time scheduling algorithm). This impacts the types of algorithms the RTOS may employ to schedule tasks and perform other system management operations. For example, algorithms that are non-starvation free, retry-based, or use unpredictable heuristics, may be acceptable in a general purpose OS, but are rarely so in an RTOS. Real-world constraints sometimes make priority inversions unavoidable. However, techniques can be used to mitigate their negative effects on timeliness. For example, in Section 2.2.3, we investigate strategies that minimize unavoidable priority inversions due to device interrupt handling.

The second requirement above may be harder to achieve by an RTOS, since application behavior partly depends upon the application itself. However, an RTOS may work in *support* of predictable behavior. For instance, an RTOS may provide predictable mechanisms for coordinated access to shared resources (*i.e.*, locking protocols). The RTOS may monitor the resources consumed by a task at runtime (*e.g.*, execution time) and use budget enforcement policies to prevent tasks from exceeding provisioned resources. If prevention is not possible, the RTOS may penalize the offending task by denying future resources, and/or use techniques that isolate the effects over-consumption may have on the rest of the system.

In this brief section, we have discussed RTOSs at only a high-level. We discuss our specific needs of an RTOS in support of GPGPU real-time systems next.

### 2.2.2 LITMUS<sup>RT</sup>

A major contribution of this dissertation is the design and implementation of real-time systems that support the matrix of CPU/GPU organizational choices we discussed in Chapter 1 (see Figure 1.4). Unfortunately, as Brandenburg (2011b) reports in a comprehensive survey of modern RTOSs, most RTOSs are limited to fixed-priority scheduling. Further, these RTOSs lack robust support for resource sharing through multiprocessor real-time locking protocols, such as those we discussed in Section 2.1.7. As we shall see in Chapter 3, we require a more feature-rich RTOS to fully explore the CPU/GPU configuration options depicted in Figure 1.4.

The GPU scheduling framework we present in Chapter 3 requires tight integration with the RTOS scheduler. As a result, most of our framework is realized by extensions and modifications to an RTOS kernel—specifically, the LITMUS<sup>RT</sup> kernel. LITMUS<sup>RT</sup> (**L**inux **T**estbed for **M**ultiprocessor **S**cheduling in **R**eal-**T**ime systems), is an open-source real-time extension to Linux which has a long development history, beginning in 2006 (Calandrino *et al.*, 2006), and remains under continual development (Brandenburg, 2011b, 2014b). LITMUS<sup>RT</sup> provides a plugin-based architecture that facilitates the implementation of, and experimentation with, multiprocessor real-time scheduling algorithms and locking protocols. It includes support for fixed-priority and deadline-based schedulers, among others. The flexibility of LITMUS<sup>RT</sup> enables us to explore a variety of CPU/GPU configuration.

The use of a Linux-based operating system such as LITMUS<sup>RT</sup> is critical to the research of this dissertation, as it allows us to leverage support of GPGPU technology (*e.g.*, GPU device drivers and GPGPU runtime (described in Section 2.4.1)) in LITMUS<sup>RT</sup>. Moreover, high-performance GPUs capable of supporting modern GPGPU applications are primarily limited to Windows, Mac, and Linux-based platforms. The open-source nature of Linux grants us the ability to implement and effectively evaluate operating-system-level algorithms for GPU-enabled real-time systems.

LITMUS<sup>RT</sup>, being based on mainline Linux, cannot be used to host “true” (safety-critical) HRT workloads. However, true HRT constraints are problematic in a GPU-enabled real-time system anyway due to hardware complexity, the closed-source nature of GPU hardware and software, and the lack of timing analysis tools for such platforms. Still, we do not believe this limitation *totally* precludes the use of GPUs in safety-related applications. In an automotive setting, for example, the reaction time of an *alert* driver is about 700ms (Green, 2000). A GPU-based automotive component may only have to react to events within such a relatively lax time

window in order to be a viably safe system. Thus, we justify research into the use of GPUs in safety-related applications in two ways. First, we expect that algorithms developed in LITMUS<sup>RT</sup> are transferable to HRT operating systems in the future. Second, even if GPU technology cannot support HRT requirements for GPU-related computations, GPUs may still be utilized in real-time systems with mixed hard and soft requirements, also known as multi-criticality systems (Vestal, 2007), as long as SRT GPU-related operations do not interfere with HRT CPU computations.

### 2.2.3 Interrupt Handling

Interrupts are a hardware signaling mechanism that may be used by asynchronously operating computing elements (*e.g.*, processors and peripheral devices) to communicate with one another. An interrupt communicates the occurrence of an event. For example, a network card may raise an interrupt to signal the arrival of a network packet—such an interrupt is received (or handled) by system CPUs. Interrupts may also be transmitted among CPUs (*i.e.*, IPIs). IPIs can be used to coordinate scheduling in a multiprocessor system. Interrupts may also be used by high-resolution timing hardware to signal the expiry of a timer. These timers may be leveraged to realize accurately timed job releases and precise budget enforcement features in event-driven real-time schedulers.

In the case of device management, device drivers may bind, or register, an interrupt handler to a uniquely identified interrupt, or interrupt line. A device driver may multiplex multiple events on one interrupt line. Moreover, interrupt lines may sometimes be shared by multiple devices. In such cases, multiple interrupt handlers may be executed in sequence upon receipt of an interrupt from a shared interrupt line.

Upon receipt of an interrupt, a CPU halts its currently executing task and invokes an interrupt handler, which is a segment of code responsible for taking the appropriate actions to process the interrupt. An interrupted task can only resume execution after the interrupt handler has completed. The CPU may be prevented from handling other interrupts during this time as well. Interrupt handlers must execute quickly so that the interrupted task can resume execution, and so the CPU can be responsive to other interrupts.

Interrupts require careful implementation and analysis in real-time systems. In uniprocessor and partitioned multiprocessor systems, an interrupt handler can be modeled as the highest-priority real-time task (Jeffay and Stone, 1993; Liu, 2000), though the unpredictable nature of interrupts in some applications may require conservative analysis. Such approaches can be extended to multiprocessor systems where tasks may migrate between CPUs (Brandenburg *et al.*, 2010). However, in such systems, the subtle difference

between an interruption and preemption creates an additional concern: an interrupted task cannot migrate to another CPU. As a result, conservative analysis must also be used when accounting for interrupts in these systems as well. A real-time system, both in analysis and in practice, benefits greatly by minimizing interruption durations. Split interrupt handling is a common way of achieving this, even in non-real-time systems.

Under *split interrupt handling*, an interrupt handler performs the minimum amount of processing necessary to ensure proper functioning of hardware. This may include an acknowledgement of receipt and any processing needed to identify the context of the interrupt (*i.e.*, demultiplexing). We call this immediate processing the interrupt handler’s *top-half*. Additional work to be carried out in response to an interrupt is deferred.<sup>14</sup> We call this processing the interrupt handler’s *bottom-half*. Bottom-half computations are executed at a more “opportune” time. However, each operating system may have a different notion of when that opportune moment may be. We now discuss the implications of different interpretations.

### 2.2.3.1 Linux

We now review how Linux performs split interrupt handling. Despite its general-purpose origins, variants of Linux are widely used in supporting real-time workloads.

During the initialization of the Linux kernel, device drivers register interrupt handlers with the kernel’s interrupt services layer, mapping interrupt lines to *interrupt service routines* (ISRs)—ISRs are equivalent to top-halves. By default, any CPU may receive a device interrupt, though CPUs may be later “shielded” individually from specified interrupts through interrupt masks.

Upon receipt of an interrupt on a CPU, Linux immediately invokes the registered ISR(s). The ISR(s) are executed within the “interrupt context,” meaning that the receipt of other interrupts is disabled. Deferred work is issued by the ISRs in the form of a *softirq*, or *tasklet* (the terms are commonly used interchangeably). The *softirq* is equivalent to an interrupt handler’s bottom-half. A pending *softirq* is enqueued on one of several per-CPU FIFO queues, depending upon the source of the *softirq*. The number and designation of these queues may vary with each kernel version. However, long-established queues include (in order highest to lowest priority): `HI_SOFTIRQ`, `NET_TX_SOFTIRQ`, `NET_RX_SOFTIRQ`, and `TASKLET_SOFTIRQ`.

---

<sup>14</sup>Some interrupts can be, or may need to be, handled entirely within the top-half processing. This includes relatively lightweight handlers for IPIs.



The Linux kernel executes softirqs using a heuristic. Immediately after executing a top-half, but before exiting the interrupt context and resuming execution of the interrupted task, the kernel executes up to ten softirqs. These softirqs are taken from the softirq queues, in order of priority. For example, all network-related softirqs in the `NET_TX_SOFTIRQ` queue are processed before any in the `TASKLET_SOFTIRQ` queue. Any remaining softirqs are dispatched to one of several (per-CPU) kernel threads dedicated to softirq processing; these are the “ksoftirq” daemons. The ksoftirq daemons are scheduled with high priority, but are preemptible.

The Linux kernel is optimized for throughput. Excessive thread context-switch overheads are avoided by executing a batch of tasklets before returning from a top-half. The FIFO queue structures also bias the processing of some types of softirqs over others. However, this batch processing may introduce long interrupt latencies, leading one to wonder if this can even be considered a split interrupt system. The original motivation behind split interrupt handling is to minimize the duration of top-half execution, not *extend* this duration with additional work. Moreover, in all likelihood, in a system experiencing few interrupts (though it may still be heavily utilized), for every top-half that yields a tasklet (bottom-half), that tasklet will subsequently be executed before the interrupted task is restored to the CPU. This essentially fuses the split top-half and bottom-half into one non-split interrupt handler.

How does Linux’s softirq processing affect real-time analysis? It is generally impossible to model Linux’s interrupt processing mechanisms. Even with a model of interrupt arrival patterns, it is difficult to predict the delay experienced by an interrupted task since we do not know which or how many softirqs may be processed before the interrupt handler returns. Moreover, if a bottom-half is deferred to a ksoftirq daemon, it is generally not possible to analytically bound the length of the deferral since these daemons are not scheduled with real-time priorities. Thus, we cannot predict how long a task may wait for a given bottom-half to be processed.

Schedulability analysis under Linux is further complicated by its software architecture. Because each softirq *might* execute within the interrupt context, softirq code may never suspend—it may never block on I/O or utilize suspension-based synchronization mechanisms. If such processing is necessary, then the bottom-half may defer additional work in yet another form. Specifically, a work item dispatched to one of Linux’s per-CPU *kworker daemons*. The kworker daemons process deferred work, much like the ksoftirq daemons, but allow work items to suspend. Also like the ksoftirq daemon, kworker threads are not scheduled with real-time priorities, implying the same challenges to realizing real-time predictability.

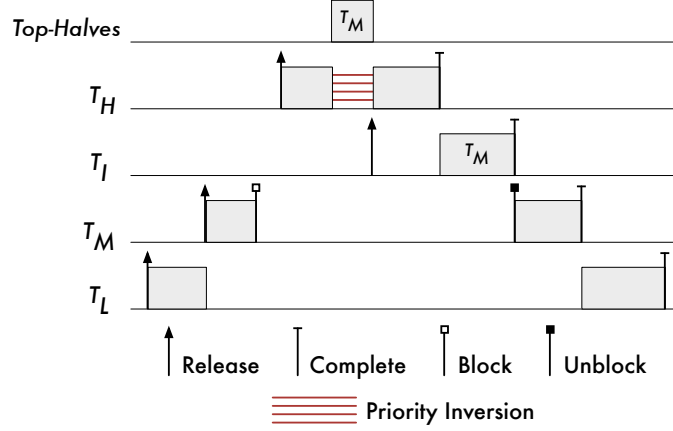


Figure 2.13: Fixed-priority assignment when an I/O device is used by a single thread.

### 2.2.3.2 PREEMPT\_RT

The PREEMPT\_RT patch to the Linux kernel alters the interrupt handling mechanisms of Linux to reduce interrupt latency and improve real-time performance. PREEMPT\_RT executes all deferred softirqs in per-interrupt source (*e.g.*, per-device) threads. Only performance-sensitive softirqs, such as high-resolution timers, may be appended to top-half execution, as this avoids thread context switch overheads. A system designer may assign an appropriate fixed priority to each softirq-handling thread, according to their application needs. For example, disk I/O softirqs can be given a lower priority than softirqs from a GPU. Under vanilla Linux, softirqs from both devices are processed with the same priority since these softirqs are put in the TASKLET\_SOFTIRQ FIFO queue.

There are scenarios where the priority assignment method of PREEMPT\_RT for interrupt handling threads is sufficient. Consider a uniprocessor real-time system with three independent threads,  $T_H$ ,  $T_M$ , and  $T_L$ .  $T_H$  is assigned a high priority,  $T_M$  is assigned a middle priority, and  $T_L$  is assigned a low priority. Suppose  $T_M$  issues a command to an I/O device and suspends from execution until an interrupt from the device, indicating completion of the operation, has been processed.  $T_M$  cannot resume execution until the bottom-half of the interrupt has completed. What priority should be assigned to the interrupt handling thread, denoted  $T_I$ , that will do this work? In order to avoid interference with other threads,  $T_I$  must have any priority greater than or equal to  $T_M$  but less than the priority of  $T_H$ . (To avoid ambiguity in scheduling, interrupt handling threads are commonly given a priority slightly greater than their dependent threads.) We refer to the priority of a task  $T_i$  with the function  $prio(T_i)$ . As depicted in Figure 2.13, with the priority assignment

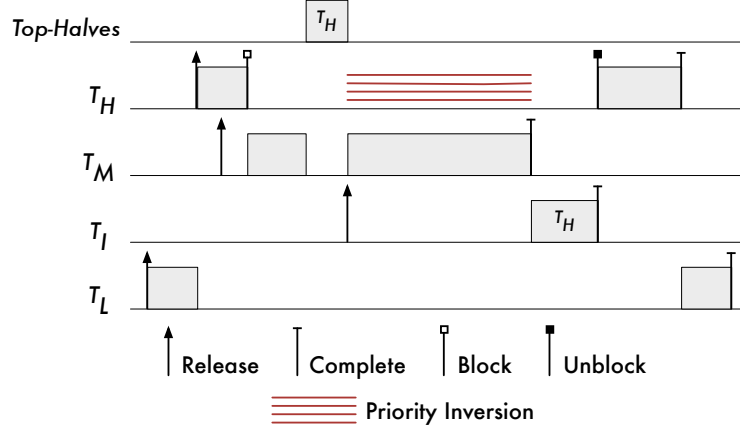


Figure 2.14:  $T_H$  may suffer a long priority inversion dependent upon the execution time of  $T_M$  if  $prio(T_I)$  is too low.

$prio(T_H) > prio(T_I) > prio(T_M) > prio(T_L)$ , the operations of  $T_M$  have less impact on  $T_H$  (priority inversions due to top-halves are unavoidable);  $T_M$  and  $T_I$  only receive processing time when  $T_H$  is not ready to run. Likewise,  $T_L$  can in no way delay the execution of  $T_I$ . Because of the lack of interference, this system is also easy to analyze. However, the situation changes when the I/O device is shared by different threads of differing priorities.

Let us reconsider the prior scenario with one change: suppose  $T_H$  and  $T_L$  share the I/O device simultaneously, and  $T_M$  does not use the device at all. Does this change the priority that should be assigned to  $T_I$ ? Indeed it does. If the priority of  $T_I$  is less than  $T_M$ , then  $T_H$  can experience needlessly long priority inversions. For example, this may occur when  $T_H$  suspends from execution after issuing a command to the I/O device and suspends to wait for completion of the command. The interrupt indicating that the operation has completed may be received, top-half executed, and bottom-half deferred to  $T_I$ , but if  $prio(T_I) < prio(T_M)$  and  $T_M$  is scheduled, then  $T_I$  cannot execute and unblock  $T_H$  until  $T_M$  gives up the processor. Thus,  $T_H$  indirectly suffers a priority inversion with respect to  $T_M$ . Such a scenario is illustrated in Figure 2.14. *Observe that the duration of this inversion largely is not dependent upon the time it takes to execute the interrupt bottom-half, but rather upon the duration between when  $T_I$  is ready to run and  $T_M$  relinquishes the processor. This dependency can break analysis and real-time predictability may not be ensured.*

The potential for such long priority inversions forces an alternative priority assignment where  $T_I$  is assigned a priority great enough to ensure  $T_H$  cannot suffer this particular priority inversion. In general, the priority of  $T_I$  must be no less than the highest-priority thread that may depend upon  $T_I$ . However, this

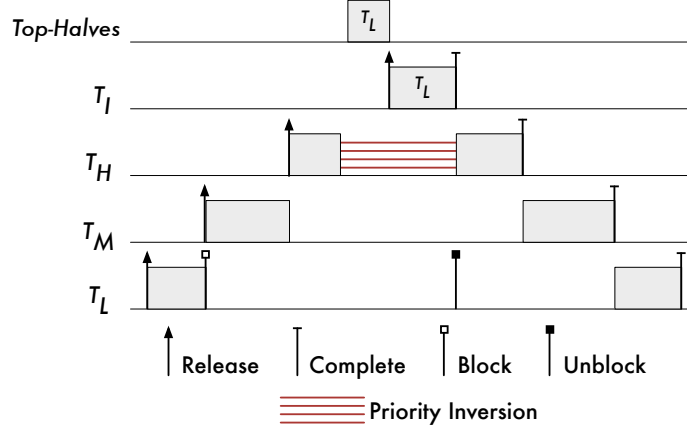


Figure 2.15:  $T_H$  may suffer a priority inversion when  $T_I$  processes a bottom-half for  $T_L$ .

assignment introduces a different priority inversion scenario. What happens when  $T_I$  processes a bottom-half for which  $T_L$  blocks, as depicted in Figure 2.15? Since  $T_I$  has the greatest priority, it is immediately scheduled whenever it is ready to run, so the bottom-half for  $T_L$  is immediately processed, resulting in the preemption of any other threads, including  $T_H$ . This is another priority inversion from which any threads with priorities less than  $T_I$ , but greater than  $T_L$ , may suffer. The primary advantage to using a greater priority for  $T_I$  is that at least these inversions are short—one inversion only lasts as long as the execution time of one bottom-half. However, the priority assignment that we have been forced to use is susceptible to pathological cases. Suppose that  $T_H$  rarely uses the I/O device and  $T_L$  uses it very frequently, generating many interrupts. Or, suppose there are *many* low-priority threads (*e.g.*,  $T_{L1}, \dots, T_{Ln}$ ) that use the I/O device. In either case,  $T_H$  and  $T_M$  may experience many priority inversions, as illustrated in Figure 2.16.

Pathological cases are undesirable and become harder to avoid when many tasks of different priorities share devices. Further, determination of a safe priority assignment for each interrupt handling thread becomes increasingly difficult with additional interrupt sources.

### 2.2.3.3 Towards Ideal Real-Time Bottom-Half Scheduling

In this section, we discuss other approaches to real-time scheduling of interrupt bottom-halves. Ideally, all bottom-halves should be explicitly scheduled in accordance with the analytical model used to determine schedulability. Under the sporadic task model, bottom-half processing should be accounted for through the addition of dedicated bottom-half processing sporadic tasks, or by somehow delegating the processing to tasks already within the task model.

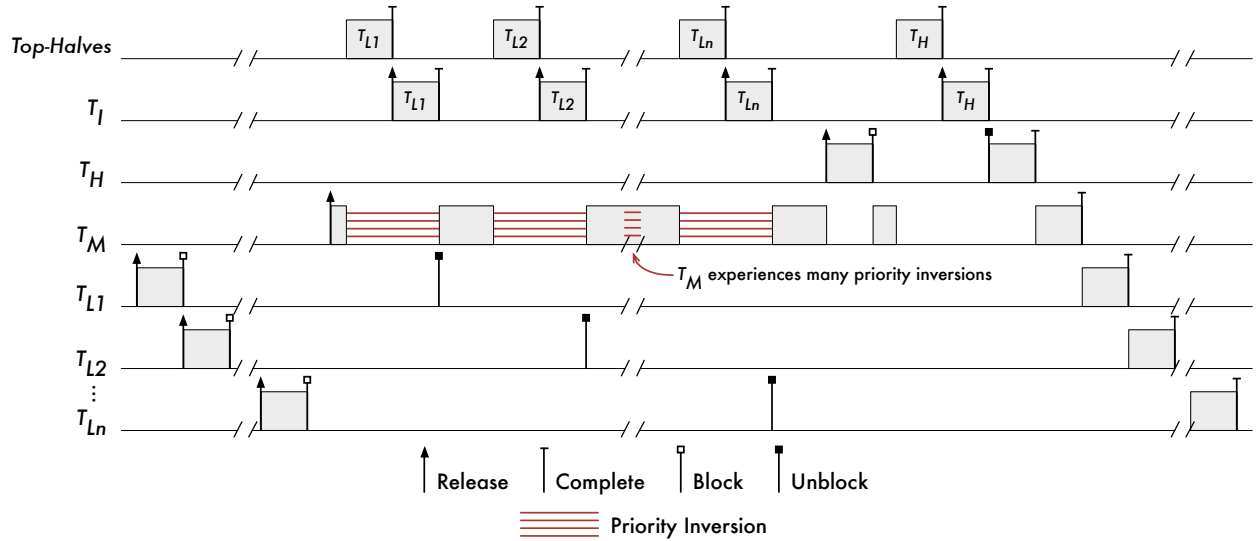


Figure 2.16: A pathological scenario for fixed-priority interrupt handling.

**Dedicated Bottom-Half Servers.** Although PREEMPT\_RT uses dedicated threads to process bottom-halves, they are not sporadic tasks. This leads to pessimistic analysis that must account for scenarios like that depicted in Figure 2.16. To force bottom-half processing to better conform to the sporadic task model, Lewandowski *et al.* (2007) and Manica *et al.* (2010) have employed dedicated sporadic tasks to handle bottom-half processing in fixed-priority and deadline scheduled systems, respectively. These dedicated tasks take the form of *servers* that receive a fixed amount of execution-time budget that is replenished periodically. This approach works well for handling bottom-halves spawned by interrupts raised in response to *external* events, such as the arrival of network packets—budgetary constraints ensure that the system is not overwhelmed by events that are outside of its control.

**Individually Prioritized Bottom-Halves.** The usefulness of the server-based approach is limited in situations where a real-time task may block while waiting for an *internal* event to occur, such as when a task waits for a GPU kernel to complete. This is because the time that the task is blocked depends upon the budget and replenishment rate of the bottom-half-processing server. An alternative approach is to *individually prioritize* and schedule each bottom-half with the priority of the task that is blocked waiting for the bottom-half to be processed. This approach is attractive because the bottom-half execution time can be analytically incorporated into the execution time of the waiting task itself. The operating system can determine a proper priority for a bottom-half upon its arrival, provided it tracks every task that currently waits for a bottom-half from a given device. Bottom-half-processing threads may *dynamically* inherit the priority of each bottom-half as it is

processed. This technique resolves the pathological case depicted in Figure 2.16, as  $T_I$  would only preempt  $T_M$  when  $T_I$  processes bottom-halves while  $T_H$  is waiting.

Individual prioritization of bottom-halves is employed by the commercial RTOS QNX Neutrino, which has a microkernel architecture. Device drivers are implemented as threaded “servers” (not to confused with the servers employed by Lewandowski *et al.* (2007) or Manica *et al.* (2010)). Servers receive and execute I/O requests from clients and also perform bottom-half processing. Characteristic to microkernel designs, clients and servers communicate through message passing channels. Device drivers receive requests for I/O operations as messages. In-bound messages are queued, in priority order, in the event that they are sent faster than they can be serviced by the device driver.

In order to avoid priority inversions, device drivers inherit the in-bound message’s priority, which is attached by the sender, when it is *sent*. The device driver inherits the maximum priority among queued, and currently processing, messages. In addition to priority, device drivers also inherit the execution time budget of their clients (a mechanism commonly referred to as “bandwidth inheritance”). This allows for the throttling of I/O workloads on a per-client basis.

Bottom-halves are delivered to the device driver for handling as event messages, and processed at the priority inherited from I/O request messages. Academic microkernels Credo (Steinberg *et al.*, 2005), an L4 extension, and NOVA (Steinberg *et al.*, 2010), a microhypervisor, have employed similar techniques to QNX Neutrino.

The benefits of threaded interrupt handling comes at the cost of additional thread context-switch overheads. (Recall that the primary reason behind Linux’s interrupt handling mechanisms to avoid these overheads.) To address these concerns, Zhang and West (2006) developed a “process-aware interrupt” (PAI) method, which supports individually prioritized bottom-halves. Here, newly spawned bottom-halves are scheduled immediately (before the interrupt top-half returns control to the interrupted task) if the bottom-half has sufficient priority. Otherwise, the bottom-half is deferred, but it is not processed by a dedicated thread. Instead, the scheduling of bottom-half processing takes place within the context-switch code path of the operating system. Prior to a context switch, the priority of the highest-priority deferred bottom-half is compared against that of the next thread to be scheduled on the processor. The context switch is skipped if the bottom-half has greater priority, and the bottom-half is scheduled instead. The bottom-half temporarily uses the program stack of the task that was scheduled prior to the aborted context switch. The resumption of

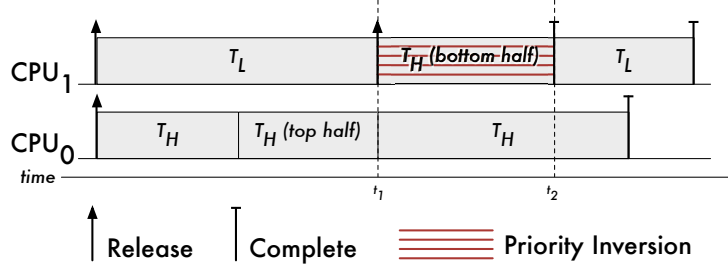


Figure 2.17: Priority inversion due to the co-scheduling of a bottom-half.

this task can be delayed since it may not be rescheduled until the bottom-half has completed, so the risk of priority inversions is not completely avoided.

**Asynchronous I/O and Multiprocessors.** The above techniques provide better real-time properties than Linux or PREEMPT\_RT, but they are not without their limitations. Thus far we have discussed device interrupts within the context of *synchronous* operations. Here, a device-using task blocks until an operation completes, as signaled by the completion of a bottom-half. However, interrupts are used in *asynchronous* operations as well, where a task may issue an operation to a device and continue execution, perhaps blocking for the operation to complete at a later time. Even with individual bottom-half prioritization, there is a risk of priority inversions under global and clustered multiprocessors schedulers.

Most real-time analysis techniques assume single threaded workload models. As such, a thread that has its priority inherited by another should never be scheduled simultaneously with that inheriting thread. Otherwise, two threads may be scheduled at the same time under the same “identity” and the non-inheriting thread analytically becomes multi-threaded, breaking analytical assumptions. Thus, it may not be correct to schedule a bottom-half of an asynchronous operation with the priority of the dependent task. The danger here is illustrated in Figure 2.17, where  $T_L$  suffers from a priority inversion within the time interval  $[t_1, t_2]$  on CPU<sub>1</sub>, when it is preempted by the bottom-half of  $T_H$ . The inversion is due to the fact that  $T_H$  is already scheduled on CPU<sub>0</sub>. We present a solution to this problem in Chapter 3, where the individual priority of a bottom-half is conditioned on the state of its dependent task.

**Engineering Challenges.** We conclude this section with the remark that dynamically tracking tasks that are blocked upon the completion of a bottom-half represents a software engineering challenge. QNX Neutrino, Credo, and NOVA overcome this challenge in part by using a microkernel architecture, whose message-passing-based architecture eases priority tracking. The same cannot be said for monolithic kernels, such

as Linux, especially where closed-source device drivers are concerned. We show how this is overcome by GPUSync in Chapter 3.

## 2.3 Review of Embedded GPUs and Accelerators

In this section, we examine the current state of GPU technology in embedded applications; this review includes a discussion of similar GPU-like accelerator technologies. This review demonstrates that there is a market for data-parallel processor architectures in embedded systems, a domain where real-time constraints are common. This confirms the relevance of the research presented in this dissertation.

GPUs may be “discrete” or “integrated.” Discrete GPUs (dGPUs) are those that plug into a host system as a daughter card. Integrated GPUs (iGPUs) are found on the same physical chip as CPUs. iGPUs are common to system-on-chip (SoC) processors targeted to embedded applications, including smartphones and tablets. dGPUs are traditionally far more computationally capable than iGPUs. It is feasible to use a dGPU in an embedded system, as long as the host platform supports the necessary I/O peripheral interconnects (*e.g.*, PCIe). Unfortunately, conventional dGPUs may not be well-suited to all embedded applications for several reasons. First, dGPUs may be physically too large, taking up too much space. Second, dGPUs commonly draw enough power (commonly between 150 watts to 250 watts) that they must rely upon active cooling, which requires a fan and an unobstructed airflow. Third, the physical port where the dGPU connects to the host system may be prone to physical vibration. However, these challenges are not insurmountable.

General Electric (General Electric, 2011) manufactures “ruggedized” dGPU platforms designed to deal with harsh embedded environments. These dGPUs may be secured to the host computing platform through reenforced I/O ports or soldered directly onto the motherboard of the computing system. Special heat-dissipating enclosures cool the GPU without the need for free-flowing air. General Electric’s ruggedized systems have been used in radar for unmanned aerial vehicles (Pilaud, 2012), sonar for unmanned underwater vehicles (Keller, 2013), and situational awareness applications in manned armored vehicles (McMurray, 2011). However, these dGPUs may still not meet the needs of every embedded application due to several limitations: (i) the heat-dissipating enclosures are large and heavy; (ii) ruggedized dGPUs draw the same power as conventional counterparts; and (iii) they are expensive. General Electric’s platforms are clearly meant for defense applications. What is affordable for a multi-million dollar military vehicle may not be affordable for a mass-market automobile.



GPU Name	Designer / Maker	GFLOPS (single-precision)	GPGPU Runtime	SoCs (not exhaustive)
GC2000	Vivante	32 <sup>a</sup>	OpenCL 1.2 (embedded)	Freescall i.MX6
SGX544 MP3	PowerVR	51 <sup>b</sup>	OpenCL 1.1	MediaTek MT6589
GC4000	Vivante	64 <sup>a</sup>	OpenCL 1.2 (embedded)	hiSilicon K3V2
Mali-628 MP6	ARM	109 <sup>c</sup>	OpenCL 1.1	Samsung Exynos 5422
G6400	PowerVR	256 <sup>d</sup>	OpenCL 1.2	Renesas R-Car H2
GC7000	Vivante	256 <sup>a</sup>	OpenCL 1.2	—
Radeon HD 8210	AMD	256 <sup>e</sup>	OpenCL 1.2	AMD A4-1340
HD Graphics 4000	Intel	295 <sup>f</sup>	OpenCL 1.2	Intel BayTrail-T
Mali-760 MP16	ARM	326 <sup>h</sup>	OpenCL 1.2	—
GX6650	PowerVR	384 <sup>e</sup>	OpenCL 1.2	Apple A8 (iPhone 6)
K1	NVIDIA	384 <sup>e</sup>	OpenCL 1.2, CUDA	NVIDIA Tegra K1

<sup>a</sup> Vivante (2014)

<sup>b</sup> Klug (2011)

<sup>c</sup> Sandhu (2013)

<sup>d</sup> Shimp (2013b)

<sup>e</sup> Smith (2014b)

<sup>f</sup> Shimp (2013a)

<sup>h</sup> Athow (2013); Smith (2014b)

Table 2.5: Performance and GPGPU support of several embedded GPUs.

Although less capable, iGPUs may offer a viable alternative to dGPUs for some applications. iGPUs lack the physical limitations of dGPUs. The size of an iGPU is negligible as it resides on-chip with CPUs. The interconnect between the host system and iGPU is also on-chip, so it is not prone to physical vibration. iGPUs require far less power; common SoCs with iGPUs commonly draw four watts of power, and rarely more than eight watts. As a consequence, iGPUs seldom require active cooling. In addition to these ideal physical characteristics, iGPUs are also more affordable, due to the economies of scale in the smartphone and tablet markets.

We now examine recent trends in iGPU performance and capabilities. Table 2.5 lists several recent iGPUs and their characteristics. We quantify computational capabilities in terms theoretical peak floating point performance, measured in GFLOPS. Unfortunately, GPU manufacturers do not always provide these numbers to the public. As a result, our data is gathered primarily from technology news websites. Each source is cited by footnote. We caution that this data may not be entirely precise. Nevertheless, we are confident that the GFLOPS reported in Table 2.5 are accurate enough to get a sense of performance.

We begin by observing trends in computational performance. The Freescale i.MX6, which includes the GC2000 iGPU, was first announced in early 2011. The NVIDIA Tegra K1, which includes the K1 GPU, was first made available to developers in mid-2014. We see that the K1 is twelve times faster than the GC2000 (32 versus 384 GFLOPS). The K1's performance is not unusual. The Mali-760 MP16 and the GX6650 perform at a similar level. The K1, Mali-760 MP16, and GX6650 were released in 2014. Comparing the GFLOPS of these recent iGPUs to the trends in Figure 1.1(a), we see that the performance of an iGPU today is roughly

equivalent to a high-end dGPU in 2006. We note, however, that dGPUs of that era regularly required over 150 watts of power. Contrast this with the four to eight watts of an *entire* SoC today.

In Table 2.5, we also observe wide adoption of GPGPU technology. Table 2.5 lists six different GPU designers that produce iGPUs with GPGPU support. These GPUs are licensed by even more SoC manufacturers. We see that OpenCL 1.2 is widely supported. Only the four least-performing GPUs are limited to OpenCL 1.1 or the embedded profile of OpenCL 1.2.

iGPUs that support GPGPU also cross instruction-set boundaries. Although the SoCs in Table 2.5 predominantly use the ARM instruction set, we also see support for the x86 instruction set from the Intel BayTrail-T and AMD A4-1340.

It is difficult to judge which GPUs best support an embedded real-time system, as this is not strictly defined by the GPU. Other SoC features are important to consider as well. Freescale and Renesas have an established presence in embedded markets. They have demonstrated an understanding and appreciation of real-time system constraints. In contrast, Samsung, Apple, and NVIDIA largely focus on consumer electronics like smartphones and tablets. Each tailors their SoC for their selected market. For example, the Tegra K1 includes a modern cell phone radio for smartphones and tablets (NVIDIA, 2014f). However, it lacks integrated support for CAN, a data bus commonly used in automotive electronics. The converse is true of the Renesas R-Car H2—it supports CAN, but lacks a cellphone radio (Renesas, 2013).

There are also differences in software to consider. For instance, the CUDA programming language is more succinct than OpenCL. Less code is necessary to perform the same operations. Moreover, NVIDIA has developed a broad set of tuned CUDA libraries and development tools. As a result, development may proceed faster on a K1 than it might on any of the other OpenCL-only GPUs. The instruction set of the SoC may also affect development. For example, the Intel BayTrail-T and AMD A4-1340 support the x86 instruction set. Development on these platforms benefits from a wide set of tools and software libraries originally developed for desktops and servers. Also, prototypes developed on x86 workstations are easier to port to x86 SoCs, than ARM SoCs.

Before concluding with this survey of iGPUs, we wish to discuss digital signal processors (DSPs) designed to support computer vision computations. These DSPs function much like an iGPU that executed GPU kernels. As such, we can apply the same GPU scheduling techniques we present in Chapter 3 to these DSPs.

We begin with the G2-APEX DSP developed by CogniVue (CogniVue, 2014), which CogniVue licenses to SoC manufacturers. CogniVue claims that the G2-APEX consumes only milliwatts of power, making it more power efficient than iGPUs. The company provides development tools for implementing computer vision applications, including a custom version of the popular OpenCV computer vision library (OpenCV, 2014). Freescale has licensed CogniVue technology for their own SoCs (Freescale, 2014).

The other computer vision accelerator is the IMP-X4 computer vision DSP, which is incorporated into the Renesas R-Car H2 SoC (Renesas, 2013). Like CogniVue, Renesas also distributes a custom version of OpenCV. What is unique to the R-Car H2 is that it also includes a G6400 iGPU from PowerVR. As we see in Table 2.5, the G6400 is among the more modest iGPUs. However, such deficiencies may be offset when paired with the IMP-X4. Unfortunately, we were unable to obtain benchmark information from Renesas or other sources to support this speculation.

It is clear from this survey of ruggedized dGPUs, iGPUs, and unique accelerator DSPs, that there is a market for data-parallel processor architectures in embedded systems. Solutions today range from expensive military-grade hardware, to specialized embedded DSPs, to common consumer-grade electronics.

These technologies will evolve with time. What direction will this evolution take? Industry has already signaled that we can expect CPUs and GPUs to become more tightly coupled. For instance, CUDA 6.0 (released in early 2014) introduced memory management features that automatically move data between host and GPU local memory. This eases programming because it frees the programmer from the burden of explicit memory management in their program code. A yet stronger signal for tightly coupled CPUs and GPUs comes from the development of the “Heterogeneous System Architecture” (HSA), which is backed by several industry leaders. HSA is a processor architecture where CPU and GPU memory subsystems are tightly integrated with full cache coherency (HSA Foundation, 2014). A GPU is more of a peer to CPUs in this architecture, rather than an I/O device as GPUs are today. Many of the advanced features of OpenCL 2.0 (the latest revision of the OpenCL standard) require HSA-like functionality from hardware. We speculate that CPUs and GPUs with HSA-like functionality will first come from manufacturers that design both types of processors, as they are in the best position to tightly integrate them. This includes companies such as Intel, AMD, NVIDIA, and ARM. It may take more time for makers of licensed GPU processors, such as PowerVR and Vivante. Also, although dGPUs typically lead iGPUs in functionality and performance, iGPUs are likely support HSA-like features before dGPUs.

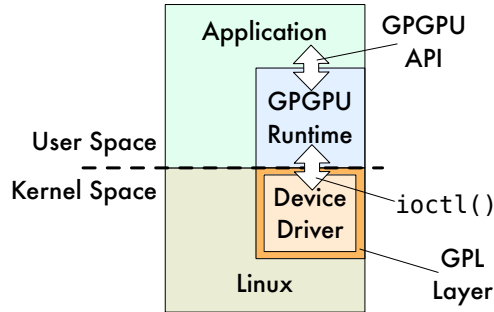


Figure 2.18: Layered GPGPU software architecture on Linux with closed-source drivers.

## 2.4 GPGPU Mechanics

In this section, we discuss: the software stack that manages GPUs; the inherently data-parallel hardware architecture of GPUs and other hardware components; and how GPUs are used in GPGPU applications.

We wish to define several terms before proceeding. The term “host” refers to CPUs and main system memory. “Device” refers to the GPU. Hence, “host memory” and “device memory” refer to main system memory and local GPU memory, respectively. The terms “process,” “application,” and “task,” all refer to a single thread of a program. “Process” carries connotations to services offered by kernel- or user-space daemons. “Application” refers to a general user-space program. “Task” refers to a program that performs a repetitive operation (real-time, or not).

### 2.4.1 Software Architecture

A complex software stack sits between a user’s GPGPU application and the GPU hardware. Figure 2.18 provides a high-level illustration of this stack in the Linux operating system. There are four primary layers: the application, the GPGPU runtime, the GPU device driver, and finally, the operating system. These layers are split across the “user space” and “kernel space” boundaries, providing the necessary memory protections between the application and operating system. We now discuss the role of each layer from top to bottom.

A GPGPU language is defined by programming language features, such as extensions to the C language, and an application programming interface (API). At runtime, application-layer code interfaces, either directly or indirectly, with the GPGPU runtime through the provided API. Direct interaction occurs when application code calls API functions explicitly. For instance, the OpenCL API `clEnqueueWriteBuffer()` is used by the application to copy data to device memory. Indirect interaction occurs when elements of the GPGPU

programming language are converted into API calls at compile-time. For example, the “triple chevrons” (*i.e.*, `<<<>>>`) used to launch a GPU kernel in the CUDA language are transformed into calls to CUDA’s `cuLaunchKernel()`.<sup>15</sup>

The GPGPU runtime manages the application’s session with the GPU. One role of the runtime is to implement GPGPU language features that do not require OS intervention. For example, both CUDA and OpenCL allow an application to attach host-side callbacks to GPU operations. These callbacks are executed by threads *internal* to the GPGPU runtime as GPU operations complete. (We discuss the implications these threads have on a real-time system shortly.)

Another function of the GPGPU runtime is to translate API calls into commands given to the GPU device driver. On Linux, these commands are passed to the driver through the `ioctl()` system call. By design, this system call does not have a strictly defined interface, as it is used by the callee to pass arbitrary data to a device driver. In this way, a device driver exposes its own API to the runtime via `ioctl()`.

The device driver is responsible for managing the GPU hardware. It communicates directly with the GPU to carry out operations requested by the GPGPU runtime. The driver may arbitrate GPU access when multiple applications wish to use the GPU at the same time. The driver is also responsible for providing device management services to the OS. These include device initialization and interrupt handling. The driver uses interfaces defined by the operating system to provide these services. This is true for even closed-source device drivers. This is depicted in Figure 2.18, where a “GPL layer” mediates the interactions between the driver and Linux. We call this the GPL layer since it bridges the closed-source driver with the Linux kernel APIs made available under the second version of the GNU General Public License (GPL) (Free Software Foundation, Inc., 1991).

For efficiency, the GPGPU runtime may also communicate *directly* to GPU hardware through a memory-mapped interface. Here, the runtime is given direct access to a segment of GPU device memory and special registers by mapping these elements into the virtual address space of the application. The GPGPU runtime can submit some commands to the GPU through this interface, bypassing the GPU driver. Similarly, the runtime can monitor the completion of commands by polling memory-mapped registers that reside on the device, instead of waiting for an interrupt raised by the device. This has obvious implications on scheduling, since neither the operating system nor the GPU driver is involved in GPU resource scheduling decisions.

---

<sup>15</sup>See line 7 in Figure 2.21 for an example of CUDA’s triple chevrons.

Unfortunately, it is not always clear which GPU operations may be issued directly or which require support from the driver. However, we do have some insights to offer. GPGPU runtimes commonly allow a CPU thread to spin or suspend while waiting for a GPU operation to complete. Our experience suggests that operation completion is actively monitored through memory-mapped registers when the user program elects to spin. Operation completion is signaled by a device interrupt when the user program elects to suspend.

The GPGPU runtime and device driver are complex software packages that are usually developed and maintained by the GPU manufacturer. Unfortunately, to date, all manufacture-produced software has been for general purpose computing, not real-time systems. This raises several issues:

1. The GPGPU API provides no mechanisms to express real-time priority or time-related parameters (such as deadlines).
2. The driver, or GPU hardware itself, may resolve contention for GPU resources using policies that are not be amenable to real-time analysis.
3. The execution time of runtime and driver operations may vary widely, with extreme outliers in worst-case behavior.
4. The software may employ synchronization techniques that break real-time scheduling.

We examine the first three issues in depth in Chapters 3 and 4. However, we further explore the remaining issue regarding synchronization now, as this provides additional insight into how the GPGPU software stack operates.

As mentioned earlier, the CUDA runtime supports the attachment of callbacks to the completion of GPU operations. Callbacks are executed by host-side threads created and managed by the CUDA runtime. These callbacks are also responsible for signaling (waking) user threads that have suspended from execution to free up CPU resources while waiting for GPU operations to complete. That is, the callback threads are used to synchronize GPU operations and user threads. This can lead to the problematic scenario depicted by the schedule in Figure 2.19 for a uniprocessor system. At time  $t_1$ , the high-priority task,  $T_H$ , suspends while waiting for a GPU operation to complete. A low-priority task  $T_L$  is scheduled at this time. The GPU operation completes some time shortly before time  $t_2$ . At time  $t_2$ , the callback thread of  $T_H$ ,  $T_H^{\text{cb}}$ , is ready to run and wake up  $T_H$ . However,  $T_H^{\text{cb}}$  was created by the real-time-oblivious CUDA runtime, so the thread lacks the priority to preempt  $T_L$ .  $T_H^{\text{cb}}$  is not scheduled until time  $t_3$ . It completes at time  $t_4$ . As a result,  $T_H$

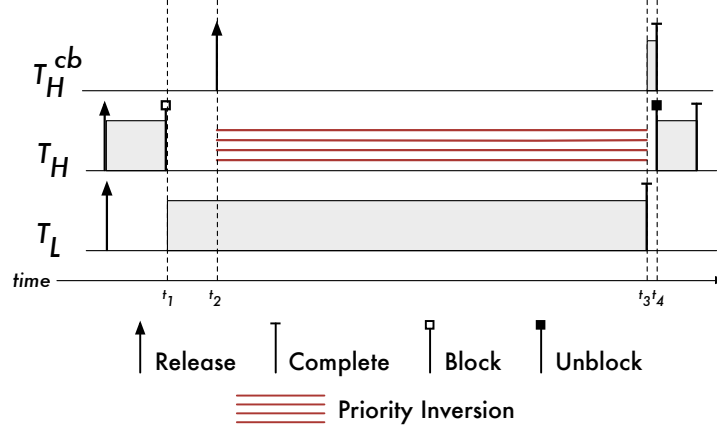


Figure 2.19:  $T_H$  experiences a priority inversion if the callback thread,  $T_H^{cb}$ , is not scheduled with a proper real-time priority.

experiences an priority inversion during the time interval  $[t_2, t_3]$ . (The inversion ends at  $t_3$  instead of  $t_4$  since  $T_H^{cb}$  performs work on behalf of  $T_H$ .) The execution time of  $T_H^{cb}$  does not affect the duration of the priority inversion—it remains constant even if  $t_4 - t_3 \approx 0$ . This scenario is not unlike the interrupt scheduling problem of Figure 2.14.

We cannot be too critical of the CUDA runtime. It was not designed with real-time scheduling in mind. The use of callback threads in dual roles of callback execution and signaling likely reduces software complexity. The approach performs well under a general purpose scheduler, since callback threads are unlikely to wait for a long time before being scheduled; such schedulers are generally responsive to threads that execute infrequently and for short durations. However, the runtime’s implementation hinders any naïve attempt at real-time scheduling.

What can we do to resolve the issues raised by the manufacturer-provided GPGPU software stack? Can it be altered? In many cases, the GPGPU software is distributed as closed-source, where direct alteration is not possible. Can it be replaced? Open-source alternatives do exist for some GPUs (*e.g.*, Gdev (Kato *et al.*, 2012) and Beignet (Segovia and Nanha, 2014)). We may alter the behavior of this software, but there are several practical issues to consider. First, the software may also be oblivious to the needs of real-time applications, despite being open-source. Second, the software may not support recent GPUs or all GPU features. Third, the software may be unable to utilize a GPU to its full potential since the software may be designed from knowledge gained through reverse engineering of GPU hardware. Finally, altered software

needs to be maintained. Replacement of manufacturer-provided GPGPU software is feasible, but it is a costly endeavor and may require the sacrifice of some functionality.

As we explore in Chapter 3, there is another approach that can be applied to both closed- and open-source software stacks: we can *wrap* the GPGPU runtime and device driver to force the software to respect real-time scheduling priorities and behave more predictably. If proven effective, such an approach should be preferred because it enables us to enjoy the benefits of manufacturer-provided GPGPU software and avoid the significant investment of alteration and maintenance of open-source alternatives.

### 2.4.2 Hardware Architecture

We describe the GPU hardware architecture in a top-down manner, beginning with how a GPU integrates into a host platform. We then discuss internal GPU components.

Recall from Section 2.3 that GPUs may be discrete (dGPUs) or integrated (iGPUs). For either type, GPUs interface to the host system as I/O devices and are managed by device drivers. Discrete GPUs differ from integrated GPUs in three ways: **(i)** they are much more computationally capable; **(ii)** they have local high-speed memory (integrated GPUs use system memory); and **(iii)** they operate most efficiently upon local GPU memory, which requires copying data to and from system memory. For most of this dissertation, we focus our attention on dGPUs for their performance characteristics and interesting challenges posed by memory management. However, the techniques developed herein remain applicable to iGPUs, except that there is no need for GPU memory management.

Figure 2.20 depicts a high-level architecture of a multicore, multi-GPU system. The GPU is connected to the host system via a full-duplex PCIe bus. PCIe is a hierarchically organized packet-switched bus with an I/O hub at its root (for this reason, the I/O hub is technically referred to as the “root complex”). Switches multiplex the bus to allow multiple devices to connect to the I/O hub. Unlike the older PCI bus, where only one device on a bus may transmit data at a time, PCIe devices can transmit data simultaneously. Traffic is arbitrated at each switch using round-robin arbitration at the packet level in case of contention.<sup>16</sup> The structure depicted in Figure 2.20 may be replicated in large-scale NUMA platforms, with CPUs and I/O hubs connected by high-speed interconnects. However, only devices that share an I/O hub may communicate directly with each other as peers.

---

<sup>16</sup>The PCIe specification allows for other arbitration schemes, but these appear to be rarely implemented (PCI-SIG, 2010).



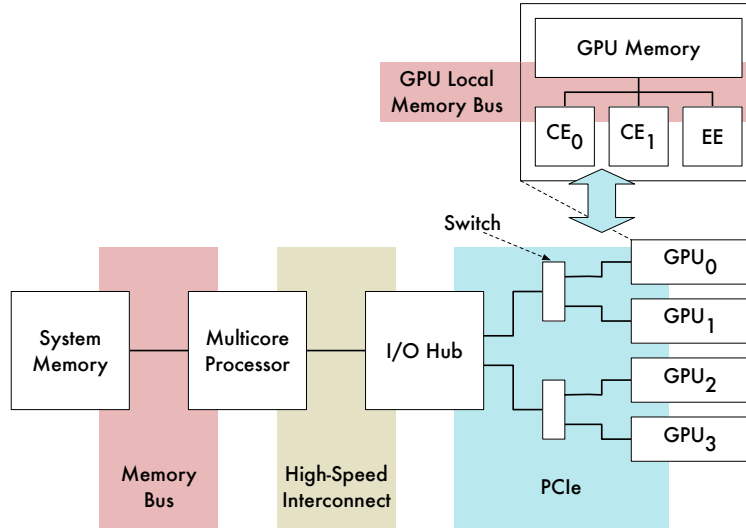


Figure 2.20: Example high-level architecture. The I/O hub may be integrated onto some multicore chips.

Within each GPU device are several specialized processors. The processors of greatest concern to this dissertation are labeled in Figure 2.20, using terminology partly defined by NVIDIA. These are the *execution engine* (EE), which is used to perform computations, and the *copy engines* (CEs), which perform bulk memory operations to interface with the host system and other I/O devices (including other GPUs). The EE and CEs share a memory bus to local GPU memory. We take a moment to address GPGPU terminology before discussing these engines in greater detail.

For the sake of consistency, we use terminology defined by NVIDIA throughout this dissertation. Our only deviation from this is the use of the term “execution engine.” NVIDIA documentation may refer to this component as the “compute engine.” We avoid this term to eschew an ambiguous abbreviation with the term “copy engine.” Although GPU architectures from different manufacturers may differ greatly, there remain many high-level similarities. Table 2.6 provides common equivalent GPGPU terminology for many of the NVIDIA-based terms we use herein. We note that because the approach we present in Chapter 3 operates by wrapping elements of the GPGPU runtime, we need not concern ourselves with lower-level differences between GPU architectures—the high-level limitations that we discuss are consistent across GPU platforms today.

<b>Software</b>	
<i>CUDA (NVIDIA)</i>	<i>OpenCL (Khronos)</i>
Thread	Work-Item
Warp	Wavefront
Block or Cooperative Thread Array (CTA)	Work-Group
Grid	NDRange
<b>Hardware</b>	
<i>NVIDIA</i>	<i>AMD</i>
CUDA Processor or Lane	Processing Element
Streaming Multiprocessor (SM)	Compute Unit (CU)
Copy Engine (CE)	DMA Engine
Compute Engine	Compute Device

Table 2.6: NVIDIA software and hardware terminology (NVIDIA, 2014c) with equivalent OpenCL (Khronos Group, 2014a) and AMD (AMD, 2013) terminology.

#### 2.4.2.1 Execution Engine

The execution engine consists of one or more parallel processors. GPU manufacturers scale the number of processors in the EE to realize GPUs with different computational capabilities. This allows the manufacturer to cover embedded, laptop, desktop, gaming enthusiast, and supercomputing markets with a common processor architecture. The number of parallel processors in a GPU also varies with each hardware architecture—especially among GPUs of different manufacturers. For example, recent NVIDIA GPUs typically have between one to sixteen processors, while AMD GPUs may have up to 44 (Smith, 2013).

For NVIDIA GPUs, each parallel processor is called a “streaming multiprocessor” (SM). Each SM is capable of executing a single instruction concurrently across several “lanes” of data operands. At any instant, a group of tightly coupled user-defined threads, called “warps,” are bound to these lanes, one thread per lane. Thus, the threads in a warp are executed in lock-step. If threads diverge on a conditional-code branch (*e.g.*, an if/else-branch), then each branch is executed in turn, with the appropriate threads “masked out” within each branch to ensure each thread executes the correct branch.<sup>17</sup>

Although an SM can only execute one warp at a time, an SM may be oversubscribed with several warps at once. That is, several warps may be assigned to a single SM. This is done to facilitate the hiding of memory latencies. The SM will quickly context switch to another ready warp if the currently executing warp stalls on a memory access.

---

<sup>17</sup>It is for this reason that GPU performance on branchy code is poor.

```

1 // Operate on 'input' of size (x, y, z) in (4, 4, 4)-sized thread blocks
2 void kernel3d_host(int ***input, int x, int y, int z)
3 {
4     dim3 size3d(x, y, z);
5     dim3 blockSize(4, 4, 4);
6     dim3 gridSize(data.x / blockSize.x, data.y / blockSize.y, data.z / blockSize.z);
7     kernel3d_gpu<<<gridSize, blockSize>>>(input, size3d);
8 }
9
10 // A 3D CUDA kernel
11 __global__
12 void kernel3d_gpu(int ***input, dim3 size3d)
13 {
14     // Compute spatial location of thread within grid
15     int i = blockDim.x * blockIdx.x + threadIdx.x;
16     int j = blockDim.y * blockIdx.y + threadIdx.y;
17     int k = blockDim.z * blockIdx.z + threadIdx.z;
18
19     // Only operate on input if thread index is within boundaries
20     if((i < size3d.x) && (j < size3d.y) && (k < size3d.z))
21     {
22         ...
23     }
24 }

```

Figure 2.21: Code fragments for a three-dimensional CUDA kernel.

To better understand how warps are assigned to the EE, we must first briefly discuss the general GPGPU programming model. In GPGPU programs, threads are organized in a hierarchical and spatial manner. Warps are groups of tightly-coupled user threads. Warps are grouped into one-, two-, or three-dimensional *blocks*. Blocks are arranged into one-, two-, or three-dimensional *grids*. One grid represents all the threads used to execute a single GPU kernel, as discussed in Chapter 1. Conceptually, it may help to think of individual threads as mapped to a single inner-most iteration of a singly-, doubly-, or triply-nested loop. At execution time, lane-specific hardware registers within the SM inform the currently executing thread of its location within its grid. Using this information, user code can properly index input and output data structures.

A simple example is illustrated by the code fragments in Figure 2.21. In line 5, the host configures a GPU kernel to process data in three-dimensional blocks of size 4x4x4 threads. The number of blocks in the grid is computed in line 6, assuming that the problem size divides evenly by four. This dimensional configuration of the kernel is provided at kernel launch, in line 7. The code in the function `kernel3d_gpu()` is programmed from the perspective of a single thread—one thread among many within the grid. This thread determines its  $(i, j, k)$  coordinates within the grid on lines 15 through 17. If the coordinates are within the bounds of the problem (line 20), then the thread operates on the input data. If the input problem has the dimensions of 128x128x128, then the resulting grid has 32x32x32 blocks, each with 4x4x4 (or 64) threads. This breakdown of the grid into threads is illustrated in Figure 2.22. Each multi-dimensional block is

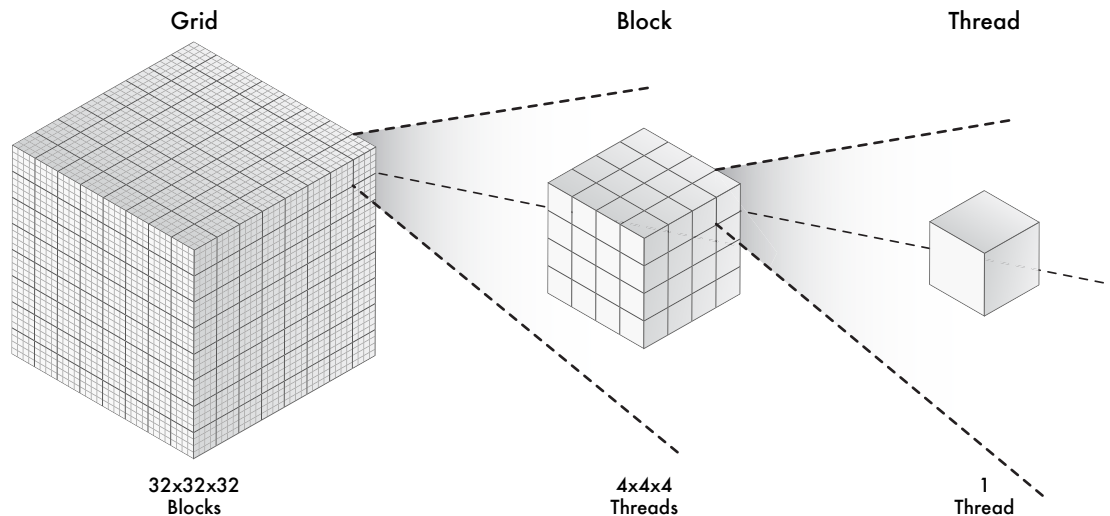


Figure 2.22: Grid of 32x32x32 blocks, with blocks of 4x4x4 threads.

linearized and decomposed into warps. The threads of each warp are mapped to the hardware lanes. Although not specified by the language, all warp sizes to date on NVIDIA GPUs have been 32 lanes, so under this assumption, each block would be made up of two warps. This is illustrated in Figure 2.23.

The EEs of modern GPUs are capable of executing more than one kernel concurrently.<sup>18</sup> This feature can be leveraged to increase EE utilization. Consider the following situation. Suppose we have several independent kernels to execute. We issue each kernel to the GPU, one at a time, waiting for each issued kernel to complete before issuing the next. Recall that grid blocks are distributed among the EE's SMs. Towards the end of the execution of each kernel, SMs will begin to idle after completing their assigned blocks, while other SMs continue executing their remaining work. At the instant before the last block completes, all but one SM will be idle. However, if we issue the independent kernels to the GPU in quick succession, *not* waiting for each issued kernel to complete before issuing the next, then SMs can be kept busy as they execute blocks of grids that have already been queued up for execution.

A GPGPU kernel is decomposed into a collection of multidimensional blocks, which are made up of threads that are grouped into warps. How is each SM assigned warps to execute? Although SMs execute the instructions of a single warp at a time, SMs are not assigned individual warps. Instead, SMs are assigned blocks, and the SMs independently schedule the warps within each block. How are blocks assigned to SMs? In today's technology, blocks are assigned to SMs by in-silicon hardware schedulers on the GPU (Bradley,

<sup>18</sup>There may be restrictions, however. For example, on NVIDIA GPUs, memory subsystems of the GPU require that concurrently running kernels share the same address space.

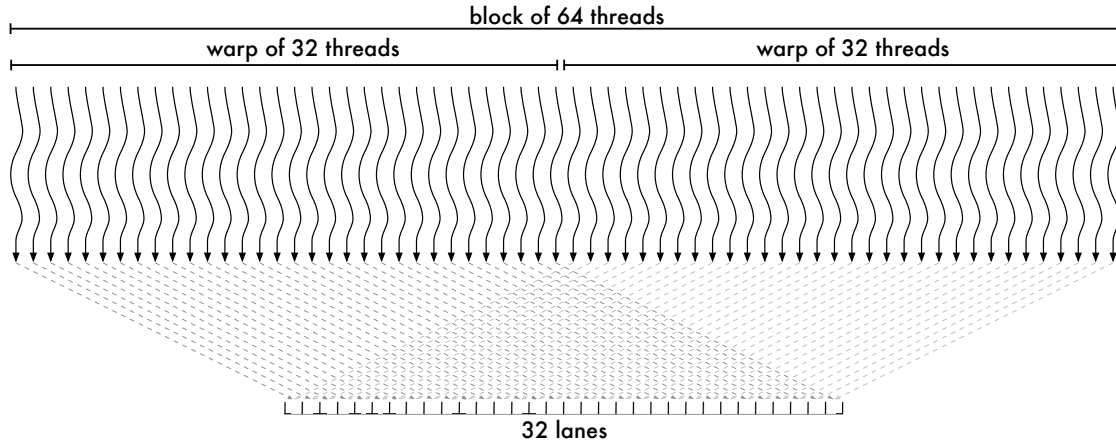


Figure 2.23: Linearized block of threads decomposed into two warps, which are multiplexed on hardware lanes.

2012). As software engineers, we have no direct control over how work is distributed among the SMs.<sup>19</sup> We merely provide the GPU with a collection of blocks, which are then scheduled by the GPU itself. This has two important implications:

1. We cannot predictably schedule individual SMs.
2. We cannot predict how SMs may be shared among concurrent kernels.

It is for these reasons that we consider the SMs together as a single processor—the execution engine—rather than individual processors.<sup>20</sup> *We can predictably schedule the EE.*

There is one more aspect of the EE that affects real-time predictability: the EE is non-preemptive. This is understandable, given the complexities of the GPU’s hardware scheduler. Non-preemption has a significant impact on any real-time system in two ways. First, it becomes impossible to strictly enforce budgets on task execution time. At best, any real-time scheduling algorithm may only attempt to avoid and *isolate* the harmful effects violations of provisioned execution times may have. Second, priority inversions become inevitable under any work-conserving scheduler. It will always be the case that low-priority work may be scheduled on an idle EE at time  $t$  when higher-priority work for the EE arrives at time  $t + \epsilon$ . The higher-priority

<sup>19</sup>This holds true for AMD GPUs as well (AMD, 2013).

<sup>20</sup>OpenCL 1.2 supports an optional feature called Device Fission. This feature allows the system designer to divide a single compute device into several logical compute devices. This may be exploited to reserve a segment of compute resources for high-priority work. However, support for this feature is currently limited to OpenCL runtimes that execute on CPUs (including Intel’s Xeon Phi) and tightly coupled heterogeneous processors, such as the Cell BE. We note that the problem of scheduling several logical devices is very similar to scheduling a multi-GPU system.

work cannot be scheduled until the low-priority work completes. We must develop algorithms that limit the duration of such inversions and account for them in analysis.

#### 2.4.2.2 Copy Engines

The GPU copy engines transmit data between local GPU memory, remote I/O memory (*e.g.*, other GPUs or I/O devices such as network cards), and system memory. The CEs operate through DMA operations. GPUs commonly have only one CE, and thus cannot send and receive data at the same time. However, high-end GPUs (such as those used in high-performance computing applications) may have an additional CE, enabling simultaneous bi-directional transmissions.

Each CE may only be tasked with one DMA operation at a time. The command to perform a DMA operation must be issued by host-side code. That is, a GPU kernel cannot issue the DMA command itself. This implies that any real-time DMA scheduling must be performed from the host.<sup>21</sup> Like EEs, CEs are also non-preemptive, incurring the same issues regarding budget enforcement and priority inversions.

DMA operations may only read or write data that is fixed, or “pinned,” to a physical memory address. This is necessary in order to prevent data from being relocated by the OS while it is acted upon by a DMA processor. GPGPU runtimes provide APIs that allow the user to pin data. However, if the user requests a DMA operation that acts upon non-pinned data, then the GPGPU runtime must take extra measures to orchestrate the operation—the particular steps depend upon the runtime implementation. For example, the runtime may automatically copy data to a staging buffer that *is* pinned and perform the DMA operation on this buffer. The operation may be performed incrementally if the staging buffer is smaller than the data accessed. This method is inefficient, since it requires data to be temporarily copied to the staging buffer. Alternatively, the runtime may dynamically pin and unpin user memory as needed. This method incurs overheads, since pinning and unpinning requires support from the OS and device driver. In real-time programming, it is already common practice to pin all application data in order to avoid page faults, which are harmful to real-time predictability. We assume that this practice is extended to all GPU-related memory using provided GPGPU runtime APIs.

Some GPUs, such as those made by NVIDIA, support peer-to-peer (P2P) DMA, where data is transmitted directly between I/O devices (such as two GPUs). This is more efficient than passing data between two

---

<sup>21</sup>Pellizzoni (2010) has explored scheduling DMA operations through the use of specialized interposition hardware that sits between the I/O device and the PCI bus. However, we consider this extreme method out of scope for this dissertation.

devices by way of a temporary buffer in main system memory. There are two restrictions to P2P DMA operations. First, the two peers must share the same I/O hub (root complex), as depicted in Figure 2.20. Second, in the case of P2P operations between two GPUs, it is unclear whether it is the CE of the sender, receiver, or both that perform(s) the DMA operation. It is partly for this reason that we make the conservative assumption that *both* CEs are utilized under our management approach described in Chapter 3. However, this requires that we coordinate the schedules of the source and destination CEs.

The efficiency of a P2P DMA operation is partly dependent upon the *distance* between GPUs. Distance is the number of links to the nearest common switch or I/O hub of two GPUs. For example, in Figure 2.20, the distance between GPU<sub>0</sub> and GPU<sub>2</sub> is two (one link to a switch, a second link to a common I/O hub). P2P DMA operations are generally more efficient over short distances, since there are fewer opportunities for bus contention.

### 2.4.3 Other Data Transfer Mechanisms

The use of CEs to transfers data to and from GPU memory is the most widely supported method of data transmission in GPGPU programming languages. However, GPUs may support other mechanisms. Fujii *et al.* (2013) have explored two such mechanisms: “GPC” and “IORW.” We describe these in turn. We conclude with a remark on unified memory models offered by recent versions of OpenCL and CUDA.

On NVIDIA GPUs, there are lightweight microcontrollers shared by clusters of SMs (or “graphics processing clusters” (GPCs)) that are capable of performing DMA operations, much like the CEs. However, this functionality is not normally exposed to the programmer. Fujii *et al.* enable microcontroller-based “GPC” DMA with custom firmware that is loaded by an open-source driver.<sup>22</sup> They found that GPC-based memory transmissions can be as much as ten-times faster than CE-based memory transmissions for data chunks of 4KB or less, with equivalent performance at roughly 16KB. The microcontroller outperforms the CE on small memory transmissions because there is less overhead in initiating the DMA operation. However, the GPC method has the significant drawback that it currently requires custom firmware, which must be a complete functional replacement for the vendor-provided firmware. The IORW approach may be a reasonable alternative if CEs cannot meet the memory transmission latency needs of an application.

---

<sup>22</sup>Our use of the term “GPC” generalizes three similar microcontroller-based methods explored by Fujii *et al.*

Under “memory-mapped I/O read and write” (IORW) data transmission, remote memory is mapped into the address space of the CPU or GPU. For instance, GPU device memory may be mapped directly into the address space of the CPUs. Host memory can also be mapped into the address space of the GPU processors. Data is transmitted automatically as load and store instructions that operate upon memory-mapped addresses are executed by CPUs or SMs. Fujii *et al.* showed that IORW can perform very fast and low-latency memory transfers, especially for data transmissions from the host to GPU memory. IORW performs well in situations where it is difficult to predict what data will be needed by a GPU kernel (*e.g.*, graph traversal algorithms). This is because it may be more efficient for the GPU to access data as-needed directly from the host, rather than use a CE to transmit the entire problem dataset to the GPU prior to kernel launch. The IORW method transmits less data in such a case. However, there are several potential drawbacks to IORW. One such drawback is that IORW proves inefficient in cases where a memory-mapped address is accessed multiple times by a remote processor. Each access incurs the relatively high penalty of transmitting data over the PCIe bus. In contrast, under the CE method, data is transmitted once over the PCIe bus and then accessed *locally*. Another drawback is that host-to-device IORW data transmissions are not always supported. Fujii *et al.* enabled this feature for NVIDIA GPUs through the use of an open-source GPU driver. We note, however, that industry *is* moving towards bi-directional IORW support. This is demonstrated by industry backing of HSA-like architectures, which we discussed at the end of Section 2.3.

The above drawbacks to IORW do not necessarily preclude its use in a real-time system. However, the following may: we cannot directly schedule memory transfers under IORW. Instead, data is transmitted as instructions are executed by a processor. This makes predicting the worst-case execution time of computations more challenging. Any such predictions are highly likely to be exceedingly pessimistic, and thus result in poor schedulability. Instruction-level interleaving of computation and memory transfers has another drawback: we expend processor time to perform memory operations that could be offloaded to DMA processors. We can achieve greater levels of system utilization by separately scheduling DMA-based memory transfers and processor execution time.

Recently, OpenCL 2.0 and CUDA 6.0 have introduced memory models (“coarse-grained shared virtual address spaces” in OpenCL 2.0 and “unified memory” in CUDA 6.0) that unify the CPU and GPU address spaces. Under these models, the GPGPU runtime and device driver coordinate to automatically transmit data between host and device memory on a page-based (4KB) granularity. However, these models are merely



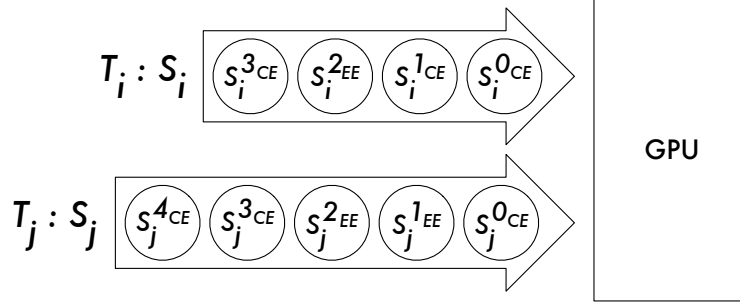


Figure 2.24: Two streams made up of sequentially ordered GPU operations issued to a GPU.

abstractions built on top of other DMA-based (*e.g.*, CE-based) mechanisms. Their use in real-time systems may be ill-advised, since the abstractions prevent the use of a real-time scheduling policy.

#### 2.4.4 Maintaining Engine Independence

The execution and copy engines of a GPU can operate independently, allowing us to schedule them separately and model them as separate processors in real-time analysis. However, limitations of in-silicon hardware schedulers that feed work to the EE and CEs can break our assumption of engine independence. We must be aware of these limitations, so that we may design around them.

GPGPU applications issue sequentially ordered GPU operations in *streams*, as depicted in Figure 2.24. Here, task  $T_i$  issues GPU operations on stream  $S_i$ , and task  $T_j$  issues GPU operations on stream  $S_j$ . We index and denote the engine used by each operation with a superscript. For example, the second GPU operation issued by  $T_i$  uses the CE and is denoted by  $s_i^{1CE}$ . (For the sake of simplicity, we assume only a single CE in this example.) New operations can be issued before prior ones have completed (*i.e.*, they may be *batched*). However, an operation may not begin execution until all prior operations issued to the same stream have completed. A single stream is somewhat analogous to a single CPU thread, as both are made up of sequentially ordered computational elements: CPU instructions for threads, GPU operations for streams.

All NVIDIA GPUs without the “Hyper-Q” stream scheduler suffer from a limitation that can break our assumption of engine independence.<sup>23</sup> On a non-Hyper-Q NVIDIA GPU, pending GPU operations of *every* stream are combined into a *single* first-in-first-out (FIFO) queue. The engine scheduler dispatches enqueued operations to the appropriate engines. However, the engine scheduler *stalls* if the GPU operation at the head of the FIFO queue has an unsatisfied stream dependency on an unfinished operation. This

<sup>23</sup>This includes all Tesla, Fermi, and non-GK110 Kepler architecture GPUs.

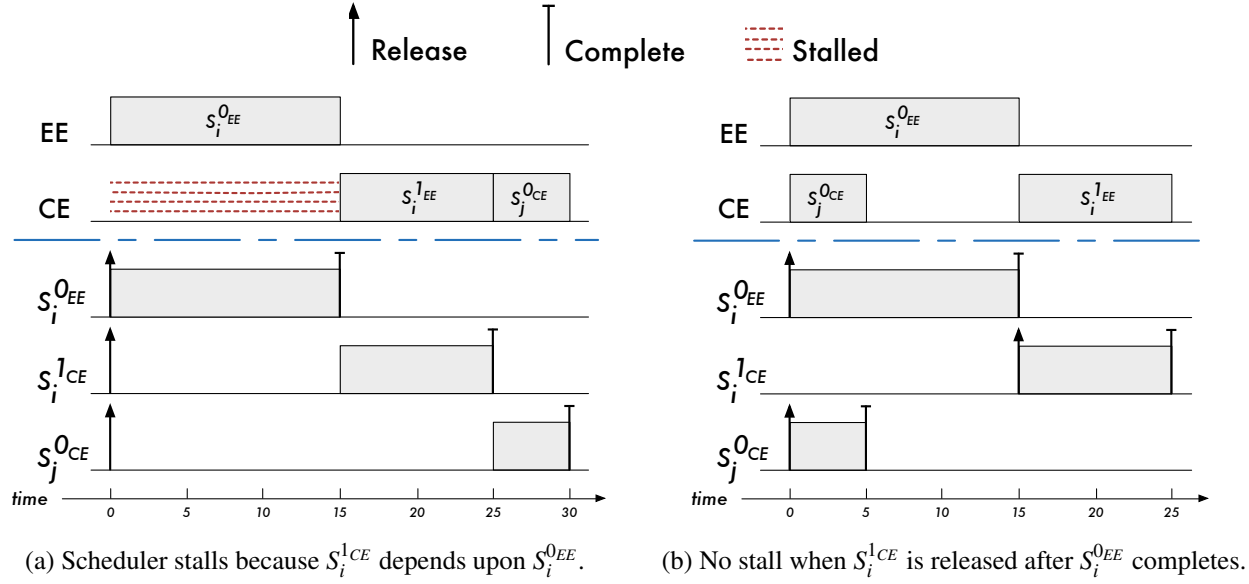


Figure 2.25: Schedules of streamed GPU operations when engines are dependent due to hardware scheduler limitations (a) and made independent through software (b).

stall prevents operations of *other* streams from being dispatched to idle engines. Such a case is illustrated in Figure 2.25 (a). Here, task  $T_i$  issues GPU operations  $S_i^{0EE}$  and  $S_i^{1CE}$  on stream  $S_i$  for the EE and CE, respectively, to a completely idle GPU. Concurrently, task  $T_j$  issues operation  $S_j^{0CE}$  for a CE on stream  $S_j$ . These operations may be enqueued in the hardware FIFO in the order:  $[S_i^{0EE}, S_i^{1CE}, S_j^{0CE}]$ . In this case, the engine scheduler dispatches  $S_i^{0EE}$  to the EE immediately.  $S_i^{1CE}$  depends upon the completion of  $S_i^{0EE}$ , so the engine scheduler stalls until  $S_i^{0EE}$  completes at time 15.  $S_i^{1CE}$  is dispatched to the CE at this time, followed by  $S_j^{0CE}$  at time 25. Observe that  $S_j^{0CE}$  is not scheduled at time 0, even though  $S_j^{0CE}$  is ready and the CE is idle—this is because  $S_j^{0CE}$  has enqueued behind  $S_i^{1CE}$  in the hardware FIFO queue. The engines are *not* independent in such scenarios. The particular order of stream operation interleaving in the FIFO queue occurs largely by chance. Observe that the illusion of engine independence would have been maintained if the concurrently dispatched operations had been ordered  $[S_i^{0EE}, S_j^{0CE}, S_i^{1CE}]$  in the hardware FIFO queue.

There are two ways to resolve the above engine-dependency issue in a real-time system. First, we may model a GPU's EE and CEs as a single notional processor. Unfortunately, such an approach results in utilization loss in schedulability analysis. For instance, suppose for a particular task set that a GPU's EE and CE each have a utilization of 51%, when the EE and CE are modeled as independent processors. Combining the EE and CE, we have a utilization of 102%. The notional processor is overutilized, so the task set is unschedulable under this analysis. An alternative approach is to design around the limitations of the hardware

scheduler: in software, each application issues GPU operations to its stream one at a time, waiting for each operation to complete before issuing the next. That is, applications “synchronize” with the GPU after issuing an operation.<sup>24</sup> It is impossible for the hardware scheduler to stall because the FIFO queue is prevented from holding more than one operation per stream. This maintains engine independence, avoiding needless utilization loss. Figure 2.25 (b) depicts the schedule for our prior example when engine independence is enforced through software. Observe that the last operation completes at time 25 instead of time 30 because the CE is used more efficiently.

Enforcing engine independence through software comes at the cost of additional overheads due to synchronization. In the case where applications suspend while waiting for GPU operations to complete, overheads include costs due to the CPU scheduler, thread context switches, and interrupt processing. These overheads must be incorporated into schedulability analysis.

#### 2.4.5 VectorAdd Revisited

In Chapter 1 (Section 1.4.1), we presented a schedule for a basic GPU kernel that adds two vectors (Figure 1.3). As we have learned in this chapter, the actual schedule is far more complex. We revisit the schedule of the `VectorAdd` routine of Chapter 1, with additional details to tie together the various operations that occur in a simple GPGPU program. Figure 2.26 depicts the schedule for our `VectorAdd` program. The upper-half of the figure depicts when various processors are scheduled. The lower-half of the figure depicts the corresponding schedule for the threads and interrupts of the GPU-using task,  $T_i$ . For simplicity, the schedule assumes that  $T_i$  executes alone on the system. Also, we assume  $T_i$  suspends from CPU execution while waiting for a GPU operation to complete. We now walk through this schedule, step-by-step.

Task  $T_i$  is scheduled on the CPU and issues a command to copy data from the host to GPU memory at time  $t_1$ .  $T_i$  suspends while waiting for this operation to complete. The copy engine  $CE_0$  is scheduled with the DMA memory copy, which completes at time  $t_2$ . The GPU issues an interrupt to the host to signal completion of the copy. Several operations occur in quick succession:

1. The interrupt handler is invoked on the CPU at time  $t_2$  and the interrupt top-half executes.

---

<sup>24</sup>Care must be taken in selecting a synchronization method. For instance, the CUDA API supports several synchronization methods: `cudaDeviceSynchronize()`, `cudaStreamSynchronize()`, and `cudaEventSynchronize()`. `cudaEventSynchronize()` cannot be used to maintain engine independence because its use actually results in the injection of a synchronization operation into the hardware FIFO queue, creating the very scenario we are trying to avoid. We recommend the use of `cudaStreamSynchronize()`, based upon our own experience.

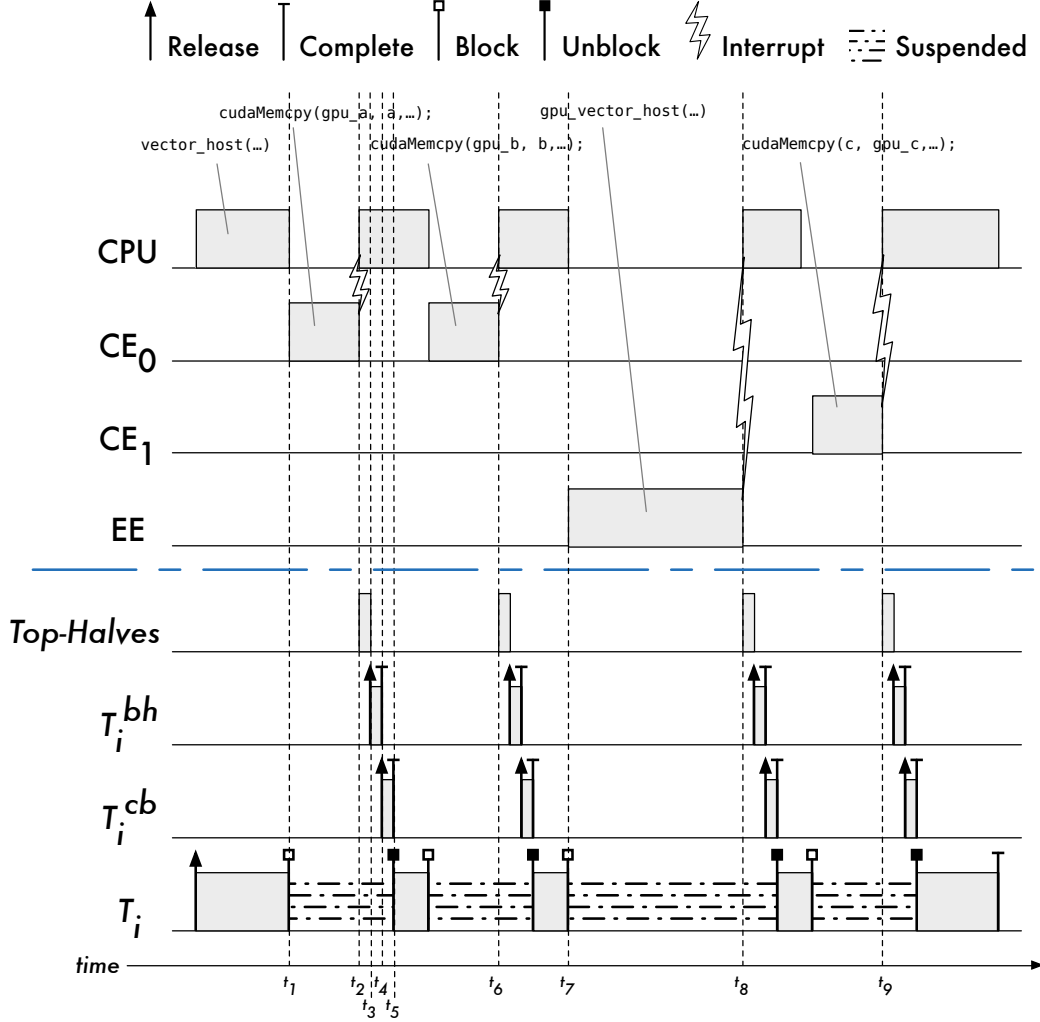


Figure 2.26: A detailed schedule depicting the interactions between the host system, GPU, and various schedulable threads, for the simple `vector_add()` kernel of Figure 1.2.

2. The top-half spawns a bottom-half, which is processed by the thread  $T_i^{bh}$ , starting at time  $t_3$ .
3. The bottom-half signals the callback thread of the GPGPU runtime,  $T_i^{cb}$ , which is awoken and scheduled at time  $t_4$ .
4.  $T_i^{cb}$  wakes the task  $T_i$  at time  $t_5$ .

This sequence of operations repeat for the remaining memory copy operations and the GPU kernel beginning at times  $t_6$ ,  $t_8$ , and  $t_9$ .

At time  $t_7$ ,  $T_i$  launches the GPU kernel. This kernel is scheduled on the EE and executes as many parallel threads across the EE's SMs. The kernel completes at time  $t_8$ . At time  $t_9$ ,  $T_i$  issues the command to copy the

results of the GPU kernel into host memory. This operation is carried out by copy engine  $CE_1$ . The copy completes and  $T_i$  finishes its execution.

The schedule depicted in Figure 2.26 should give us an appreciation for the complexity of the problems we face in developing a real-time system with GPUs. The `VectorAdd` program is simple compared to real-world GPGPU programs that may invoke many more memory copies and GPU kernels within a single real-time job. Despite this complexity, our CPU scheduler must schedule threads  $T_i^{bh}$  and  $T_i^{cb}$  with priorities no less than  $T_i$  in order to avoid priority inversions, all while respecting the priorities of other real-time tasks in the system. Access to EE,  $CE_0$ , and  $CE_1$  must be arbitrated among competing tasks with different real-time priorities. Overheads related to scheduling, context switches, and interrupt handling, to name a few, must be integrated into real-time schedulability analysis.

Figure 2.26 does not capture the full complexity of the system we propose to build. In Chapter 3, we develop multi-GPU scheduling algorithms that allow tasks to migrate among GPUs. The multi-GPU schedulers can be paired with a variety of multiprocessor CPU schedulers. Our proposed GPU scheduling framework also supports budget enforcement mechanisms that isolate the effects of occasionally poorly behaved tasks. Finally, we achieve this by wrapping the GPGPU runtime and device driver, rather than implementing our own real-time GPU software stack from the ground up.

## 2.5 Prior Work on Accelerator Scheduling

In this section, we cover prior work related to the scheduling of compute accelerators, such as GPUs. We begin with a discussion of the various directions real-time GPU scheduling has taken. We then examine prior work on scheduling computational accelerators, such as digital signal processors (DSPs) and field-programmable gate arrays (FPGAs). We conclude with brief survey of GPGPU scheduling techniques in non-real-time systems.

### 2.5.1 Real-Time GPU Scheduling

Current real-time GPU research falls within three general categories: (i) techniques for persistent low-latency GPU kernels, (ii) WCET analysis of GPU kernel code, or (iii) GPU resource scheduling. We review these categories in turn.

### 2.5.1.1 Persistent Kernels

In the first category, a persistent GPU kernel executes on a dedicated GPU. These kernels never terminate; they continually poll for work in a producer/consumer-styled software architecture. Research in this area focuses on efficient data movement between a single GPU and the rest of the system. There is no need for *scheduling* data-movement or GPU computations since there is only a single dedicated GPU and a single persistent kernel. Instead, low-latency memory operations are of primary importance. Aumiller *et al.* (2012) explored the tradeoffs between DMA memory transfers and various IORW-based data transfer mechanisms, and Fujii *et al.* (2013) explore these matters in greater depth. This work was done in support of that by Rath *et al.* (2012), wherein a GPU is used to actively adjust the magnetic field that contains plasma in a tokamak fusion reactor. Using IORW-based data transfers, Rath *et al.* (2012) report that their GPU-based solution can react to changes in input sensor data within 10  $\mu$ s. These latencies could not be achieved using DMA-based data transfers due to the overhead of setting up the CEs.

### 2.5.1.2 GPU Kernel WCET Estimation and Control

The second category of real-time GPU research has focused on bounding the execution time of GPU program code, with no attention paid to scheduling or data-movement costs—it is assumed all data already resides on the GPU. This work is useful within the context of real-time analysis. Berezovskyi has been the primary investigator in developing methods for estimating the WCET of GPU kernels. In his first effort, Berezovskyi *et al.* (2012) developed a model of GPU kernel execution in terms of the total number of lanes in an SM, the number of threads within a GPU kernel, and the number of program instructions of the GPU kernel. Under several simplifying assumptions, they developed a formulation of an integer linear program (ILP) that computes the maximal execution time of the GPU kernel across the SMs. This work matured with the development of heuristics to estimate WCET (Berezovskyi *et al.*, 2013)—this heuristic method produces a WCET estimate in several hours, instead of several days. Unfortunately, these approaches can only provide a WCET for a GPU kernel that executes on a *single* SM. This limits the scope of this work to single-SM GPUs. Thankfully, these are prevalent in iGPUs, such as those we discussed in Section 2.3.

Berezovskyi recently shifted focus towards WCET estimation through empirical measurements and statistical analysis (Berezovskyi *et al.*, 2014). This work is more practical than the prior approaches because it accounts for memory subsystem overheads (such as cache misses), as well as being applicable to multi-SM

GPUs. Further, an empirical approach towards WCET estimation is generally sound for two reasons. First, data parallel algorithms that execute efficiently on GPUs exhibit very regular memory and code path execution patterns. Second, the GPU EE largely executes in isolation—they are not burdened with an operating system, memory paging, or even multi-tasking (assuming GPU kernels do not execute concurrently on the EE).<sup>25</sup> For these reasons, deviations in observed GPU kernel execution time are relatively small in most applications, so violations of a statistically derived WCET should be rare.

Berezovskyi *et al.* are not the only ones to have researched GPU kernel WCET estimation. Betts and Donaldson (2013) have developed two WCET-estimation methods that both utilize low-level GPU kernel execution traces and the control flow graph of GPU kernel code. The first method proceeds by estimating the start-time of the last warp executed on each SM (this is derived from empirical measurements), and then estimating the execution time of the last warp. These two estimates combine to derive a final kernel WCET estimate. Under the second method, the authors model the hardware scheduler that distributes blocks among SMs. Using trace data, they estimate the execution-time costs due to inference among warps. These costs accumulate through to the final warp. Betts and Donaldson tested their methods using a cycle-accurate GPU simulator, and determined that the first method provides the most accurate results, and that the later is overly pessimistic.

WCET-estimation can be useful in hardware provisioning and used in real-time schedulability analysis. However, a WCET-estimate merely *models* system behavior—the estimate in and of itself does not guarantee that a GPU kernel will actually complete within that time. Actual enforcement of GPU kernel execution time is challenging since EEs are non-preemptive. It is possible to signal to a kernel that it *should* terminate once a provisioned WCET has been exceeded. This is accomplished through the setting of an application-defined variable (or flag) by the host, using IORW, that is checked by the GPU kernel at runtime. A kernel can voluntarily self-terminate once it detects that its “should-terminate” flag has been raised. Unfortunately, such an approach is not always practical as it can leave data in an unknown (or difficult to resume from) state. However, Mangharam and Saba (2011) explain that just such an approach is practical for “anytime” algorithms. Anytime algorithms are those that iteratively improve results as they execute; Mangharam and Saba use the parallel version of the A\* path-finding algorithm, called PAP\*, as a motivating example. A GPU kernel that implements an anytime algorithm may check its “should-terminate” flag at the top of each

---

<sup>25</sup>Contention on the GPU’s local memory bus can be a (minor) source of interference, as we show later in Chapter 4.

iteration and terminate when requested, thereby adhering to the provisioned WCET (with some measurable delay).

### 2.5.1.3 GPU Resource Scheduling

The final category of real-time GPU research is on the scheduling of GPU resources. That is, the problem of scheduling both data movement and GPU computations on GPU(s) shared by competing jobs of different priorities. Work in this area seeks to develop real-time GPU scheduling algorithms, as well as analytical models to support schedulability analysis. This dissertation falls within this category.

TimeGraph is an early approach to the real-time scheduling of modern GPUs (Kato *et al.*, 2011b). TimeGraph plugs into an open-source GPU driver, where it intercepts the GPU commands issued by GPU-using applications. TimeGraph schedules a GPU as a single processor, scheduling intercepted commands according to a configurable scheduling policy. TimeGraph supports two scheduling policies: the “high-throughput” (HT) policy, and the “predictable-response-time” (PRT) policy. The HT policy allows commands from a task to be scheduled immediately, provided that the GPU is idle, or if commands from that task are currently scheduled on the GPU and no other commands from higher-priority tasks are waiting to be scheduled. This policy promotes throughput at the risk of introducing priority inversions—the scheduling of new commands may extend the delay experienced by higher-priority commands issued soon after. The PRT policy decreases the risk of lengthy priority inversions, as new GPU commands of a task are not scheduled until all of its prior commands have completed. TimeGraph monitors the completion status of commands by plugging into the interrupt handler of the open-source device driver.

GPU-using tasks under TimeGraph may be assigned a fixed-priority and GPU utilization budget.<sup>26</sup> A task’s GPU budget is drained as it executes commands on a GPU. TimeGraph supports two budget enforcement mechanisms: “posterior enforcement” (PE) and “*a priori* enforcement” (AE). Under PE, the budgetary deficits incurred by a task’s budget-overflow is recouped by delaying further scheduling of the offending task until its budget has been replenished. The PE strategy is to *recover* from budget overruns. Under AE, TimeGraph attempts to anticipate budget exhaustion. This is done by matching the sequence of a task’s requested GPU commands against a historical record of prior-issued GPU command sequences. The historical record contains an average execution time, which is taken as a predicted execution time of

---

<sup>26</sup>TimeGraph also supports online priority assignment for graphics (*i.e.*, non-compute) applications, where the foreground application is allowed to consume additional GPU resources.



the requested commands. A task’s requested GPU commands are not eligible for scheduling until the task’s budget is sufficient to cover the predicted execution time. Thus, the AE strategy is to *avoid* budget overruns.

TimeGraph is somewhat limited in the real-time GPGPU domain. As presented by Kato *et al.* (2011b), TimeGraph is targeted to *graphics* applications, such as video games and movie players, rather than GPGPU applications. It focuses on providing a configurable quality-of-service for applications, while loosely adhering to a fixed-priority real-time task model. TimeGraph also unifies the GPU EE and CE processors into one—we discussed the negative effects on schedulability of such an approach in Section 2.4.4. TimeGraph is limited in this way because it schedules GPU commands without inspecting them to determine their function. Separate EE and CE scheduling is impossible without this inspection.

The developers of TimeGraph later developed RGEM, a real-time GPGPU scheduler (Kato *et al.*, 2011a). RGEM is similar to TimeGraph in that it also supports fixed-priority scheduling. However, RGEM operates entirely within the user-space through a user-level API. The RGEM API provides functions for issuing DMA operations and launching GPU kernels. These APIs invoke GPU scheduler routines. Because RGEM is implemented in user-space, GPU scheduler state is maintained in shared memory accessed by each GPGPU task. Tasks that are unable to be scheduled immediately on the GPU are suspended from the CPU, awaiting for a message to proceed (delivered through a POSIX message queue). Perhaps the most notable of RGEM’s contributions is how it addresses schedulability problems caused by long non-preemptive DMA operations. Here, RGEM breaks large DMA operations into smaller chunks, reducing the duration of priority inversions and thus improving schedulability.

RGEM has several advantages over TimeGraph for GPGPU applications. Unlike TimeGraph, RGEM utilizes techniques that make it amenable to schedulability analysis under rate-monotonic scheduling. Also, RGEM separately schedules a GPU’s EE and CEs. However, as presented in Kato *et al.* (2011a), RGEM provides no budget enforcement mechanisms.

The notion of breaking large non-preemptive GPU operations into smaller ones has also been explored by Basaran and Kang (2012). In addition to chunked DMA, Basaran and Kang also developed a mechanism for breaking large GPU kernels into smaller ones. Here, the kernel’s grid of thread blocks is programmatically split into smaller sub-grids that are launched as separate kernels. Unfortunately, this kernel-splitting requires developers to modify GPGPU kernel code. As we discussed in Section 2.4.2.1, threads must compute their spatial location (index) within a grid. Kernel-splitting requires the kernel code to include additional spatial offsets in the index computation. Zhong and He (2014) recently developed a method to make these offset

calculations transparent to the programmer in a framework called Kernelet. Kernelet programmatically analyzes kernel code and patches indexing calculations at runtime. However, no one has yet attempted to apply this technique in a real-time setting—Kernelet’s just-in-time patching of GPU kernel code may present a challenge to real-time analysis.

Kato *et al.* and Basaran and Kang examined GPU scheduling strictly in terms of the sporadic task model. A different approach has been taken by Verner *et al.* (2012), where GPU operations of various jobs of sporadic tasks are combined into a *batch* at runtime and scheduled jointly. Here, batches of GPU work execute in a four-stage pipeline: data aggregation, DMA data transfer from host to device memory, kernel execution, and DMA data transfer of results from device to host memory. GPU work is batched at a rate of  $\frac{1}{4}d_{\min}$ , or one quarter of the shortest relative deadline in the task set. Consecutive batches may execute concurrently, each in a different stage of the pipeline.<sup>27</sup> Verner *et al.* has continued research on batched scheduling for multi-GPU real-time systems in Verner *et al.* (2014a,b). Although their work is targeted to hard real-time systems, Verner *et al.* only consider schedulability in terms of the GPUs only. The real-time scheduling of the CPU-side GPGPU (*i.e.*, triggering DMA and launching kernels) work remains unaddressed.

Thus far, we have discussed research in GPU resource scheduling largely in terms of systems development, *i.e.*, the design and implementation of real-time GPU scheduling algorithms. Research on developing new analytical models has also been pursued. Kim *et al.* (2013) point out that conventional rate-monotonic schedulability analysis cannot be applied to task sets where tasks may self-suspend, unless suspensions are modeled as CPU execution time (*i.e.*, suspension-oblivious analysis, as discussed in Section 2.1.6.3). In order to reclaim CPU utilization that would otherwise be lost in suspension-oblivious analysis, Kim *et al.* devised a task model whereby jobs are broken into sub-jobs, along the phases of CPU and GPU execution. The challenge then becomes assigning a unique fixed priority to each sub-job. Kim *et al.* showed that the determination of an optimal priority assignment is NP-hard. They presented and evaluated several priority-assignment heuristics.

### 2.5.2 Real-Time DSPs and FPGA Scheduling

Digital signal processors are highly specialized processors optimized to provide a limited set of computational facilities. The generality of DSPs varies from processor to processor. At one end of the spectrum,

---

<sup>27</sup>For GPUs with one CE, the host-to-device DMA stage of batch  $N$  is combined with the device-to-host DMA stage of batch  $N + 2$ .

a DSP may be an application-specific integrated circuit (ASIC), capable of performing only one type of computation, and thus lack generality. A coprocessor for performing fast Fourier transforms, a common SoC DSP component, is an example of an ASIC DSP. At the other end of the spectrum, a DSP may be entirely programmable, such as the Texas Instruments C66x processor, which can be programmed with standard C code (Texas Instruments, 2013). Between these extremes are DSPs that are partially programmable, such as the computer vision-focused G2-APEX discussed earlier in Section 2.3.

DSPs sacrifice some degree of generality in order to achieve greater energy efficiency and speed. Common DSPs found on SoCs provide services for audio/video encoding and decoding, image sensor processing, and cryptography. Although these features can be realized by software that executes on CPUs (or even a GPU), a software-based approach may be slower and less energy efficient.

In most cases, DSPs execute non-preemptively. In this way, a DSP computation is not unlike a GPU kernel scheduled on the EE. Similarly, the logic fabric of FPGAs may be configured to implement DSP-like functionality. Researchers have examined the problem of scheduling non-preemptive DSPs, and similarly configured FPGAs, in real-time systems.

Gai *et al.* (2002) examined the problem of fixed-priority scheduling on a hardware platform with one CPU and one non-preemptive DSP. In their approach, ready jobs are classified as normal or DSP-using. Normal jobs only execute on the CPU, while DSP-using jobs execute on both the CPU and the DSP. Ready normal jobs enqueue on one ready queue, while ready DSP-using jobs enqueue on another. When the DSP is idle, the scheduler selects the highest-priority job among both queues to scheduler. However, when the DSP is in use, jobs in the DSP queue are ignored. Thus, the scheduler itself resolves contention for the DSP, since no DSP-using job can preempt a scheduled DSP-using job. Gai *et al.* found that their approach was preferable in terms of schedulability in comparison to a method where the DSP was treated as a shared resource, protected by the DPCP locking protocol.

Non-preemptive DSP scheduling has also been researched by Pellizzoni and Lipari (2007) for deadline-based scheduling. Pellizzoni and Lipari break DSP-using tasks into subtasks. Each subtask is designated as normal or DSP-using, just as Gai *et al.* Pellizzoni and Lipari provided a heuristic based upon simulated annealing to derive from task set parameters a relative deadline for each subtask. DSP access is arbitrated by the SRP locking protocol. Pellizzoni and Lipari developed a schedulability test to account for the inter-task dependencies among sub-tasks.

The approaches taken by Gai *et al.* (2002) and Pellizzoni and Lipari (2007) may be applied to GPU kernel scheduling on iGPUs. However, these methods are not general enough to be extended to support dGPUs, since DMA operations must also be scheduled. Also, these solutions do not address issues raised by large complex drivers or interrupt handling. DSPs are generally less complex than GPUs, so the associated drivers are similarly less complex.

### **2.5.3 Non-Real-Time GPU Scheduling**

Before concluding this chapter, we examine prior work on GPU scheduling in the non-real-time domain. This is a valuable exercise as it gives us greater insight into how GPGPU runtimes and GPU drivers can be manipulated (or even replaced) to enact a scheduling policy, be it real-time or not. Non-real-time GPU scheduling is a broad area of research, so we limit our attention to work where support for GPGPU applications is explicit. That is, we ignore work that only examines scheduling graphics applications.

In general, non-real-time GPU scheduling research may be categorized into one of three categories: (i) GPU virtualization; (ii) GPU resource maximization; and (iii) fair GPU resource sharing. Although the research goals in each area may differ, we see that researchers often apply similar techniques.

#### **2.5.3.1 GPU Virtualization**

We now discuss the topic of GPU virtualization for GPGPU applications.

Services for cloud computing are a growing market in today’s computing industry. In order to avoid the cost of purchasing and maintaining expensive data centers, companies may rent computing services from cloud providers. Through virtualization technology, customers create their own virtual machines. From the customer’s perspective, a virtual machine offers the same functionality as a physical computer. Cloud providers multiplex several virtual machines onto a single physical computer; the virtual machines run concurrently on the shared hardware.

There is interest in offering GPGPU support in cloud services. For example, Amazon offers “GPU Instance” virtual machines that support CUDA and OpenCL (Amazon, 2014). However, the tight coupling between the GPGPU runtime, GPU driver, and GPU hardware makes the multiplexing of multiple virtual machines on a GPU non-trivial. Indeed, Amazon side-steps this issue entirely—each GPU Instance virtual machine is allocated a dedicated GPU.

GPU virtualization solutions are starting to be produced by GPU manufacturers themselves. NVIDIA recently announced a product called “vGPU,” where up to eight virtual machines may share a single physical GPU (NVIDIA, 2014d). However, vGPU supports virtualization for graphics applications only. There is no support for virtualization of GPGPU services—dedicated GPUs are still required.

Researchers have developed several prototypes for virtualizing GPGPU services: GViM by Gupta *et al.* (2009), gVirtuS by Giunta *et al.* (2010), vCUDA by Shi *et al.* (2012), and GPUvm by Suzuki *et al.* (2014). Implementation details differ among these prototypes, but they all (with the exception of GPUvm) share the same RPC-styled software architecture. Here, GPGPU API calls made by processes within the virtual machines are intercepted and translated into remote procedure calls that are executed within the environment that hosts the virtual machines. API interception is handled by an API-compatible *stub library* that replaces the GPGPU runtime within each virtual machine. The stub library communicates API calls to a GPGPU backend user-space daemon that runs within the host environment, outside the purview of the virtual machines. This backend services the remote procedure calls using the full GPGPU runtime to communicate with the GPU driver and GPU. These GPGPU virtualization prototypes use different mechanisms for transmitting remote procedure call data between the virtual machines and the host environment. In order to avoid costly memory copy operations between the virtual machines and host environment, these prototypes also implement mechanisms for remapping pages of memory between the virtual machines and the host environment.

GPUvm differs from the other GPU virtualization techniques in that it is not API-driven. Instead, GPUvm presents a logical instance of a GPU to each virtual machine through the hypervisor. This is achieved by creating “shadows” of memory regions, including GPU memory addresses. This technique allows applications within the virtual machines to use standard GPGPU runtimes instead of stub libraries. A GPUvm backend user-space daemon runs the host environment and monitors access to shadowed memory addresses, shuttling GPU commands and results between the virtual machines and the physical GPU.

In the software architectures described above, GPU scheduling policies can be implemented within the backend daemons that service remote procedure requests (or GPU commands, in the case of GPUvm). For instance, requests can be serviced with a simple policy like first-come-first-serve or round-robin. More advanced policies may also be employed. GViM and GPUvm use scheduling algorithms inspired by the Xen hypervisor “credit” scheduler. Here, a budget for GPU operation execution time is assigned to each virtual machine. The appropriate budget is drained as GPU operations from the associated virtual machine

are executed. Budgets are replenished periodically. GPU resources can be fairly allocated to virtual machines by setting the appropriate budgets.

### 2.5.3.2 GPU Resource Maximization

The RPC-styled software architecture of the virtualization techniques discussed above can also be used to maximize GPU resource utilization. Recall that an EE is capable of running several GPU kernels concurrently, provided that these kernels share the same address space (see Section 2.4.2.1).<sup>28</sup> A kernel completes when the last block of its grid completes. If kernels are *not* executed concurrently, then all but one of an EE’s SMs will always be left idle while the last block executes. However, these idle SMs can be kept busy if additional kernels have been queued for concurrent execution.

By funneling all GPU operations through a single process, similar to the RPC backend in the virtualization prototypes, a greater degree of EE utilization can be achieved. rCUDA is the first attempt at implementing such a framework for the CUDA runtime (Duato *et al.*, 2010). rCUDA consists of two major components: a CUDA API-compatible stub library and an RPC backend daemon. In addition to servicing requests made by local processes, the rCUDA RPC server can also service GPU operation requests from *remote* machines in a compute cluster. VOCL provides a similar framework for the OpenCL runtime (Xiao *et al.*, 2012). Recently, NVIDIA released a framework of their own, called “Multi-Process Service” or MPS (NVIDIA, 2014b). Unlike rCUDA or VOCL, MPS lacks the ability to service requests from remote machines.

With regards to GPU scheduling, the rCUDA daemon services requests from clients in a round-robin fashion. VOCL’s daemon schedules work in first-come-first-serve order. These schedulers do not separately schedule the EE and CEs, so engines may be left idle even though there may be work ready to be scheduled on them. NVIDIA has not disclosed the scheduler employed by MPS.

We conclude with a remark relating these GPU resource maximization techniques to the virtualization methods we discussed earlier. The authors of rCUDA and VOCL describe their approaches as a solution for GPGPU virtualization. However, we opt to separate rCUDA and VOCL from the virtualization category because their software architectures match that of a conventional RPC-based distributed system. No attempt is made to take advantage of a virtualized environment. Specifically, they do not eliminate memory copies

---

<sup>28</sup>Technically speaking, in CUDA, these kernels must actually belong to the same CUDA “context.” A single context can be thought of as a GPU address space. The CUDA runtime creates one default context per GPU for each process. A single process can create multiple contexts for the same GPU using a low-level CUDA API, but this feature is rarely used.

between the client and server when they reside on the same physical machine; this could be done through memory remapping. Instead, rCUDA and VOCL transmit all data between the client and server through network sockets.<sup>29</sup>

### 2.5.3.3 Fair GPU Scheduling

We now discuss several notable works on fair GPU scheduling. We begin with PTask (Rossbach *et al.*, 2011). The PTask framework defines a set of OS-managed abstractions. These abstractions give the OS insight into an application’s various phases of execution on CPUs, EE, and CEs. This insight is leveraged by the OS to schedule GPU resources as “first class” processors (*i.e.*, exercise a degree of control commensurate to that exerted on CPUs). In the PTask framework, GPU-using tasks are decomposed into a dataflow graph. Graph *vertices* loosely represent a computation or operation on a CPU or EE. We describe this representation as “loose” because EE vertices must still be scheduled on a CPU in order to initiate GPU operations. Moreover, an EE vertex may still contain considerable CPU code, if desired by the programmer. Each vertex exposes a set of input and output *ports*. Each port is backed by a data buffer. These data buffers reside either in host memory or GPU memory. Ports are connected by *channels*, which represent the data dependencies among vertices. CE operations are deduced by examining the memory type (host or GPU) of data buffers that are connected by a channel. Connected ports with buffers of different memory types require the CE to shuttle data between them. Connected ports with buffers of the same memory types are combined into one, eliminating the need for memory copy operations.

GPU operations are initiated through a set of PTask-defined APIs that operate on the vertex, port, and channel abstractions. These APIs call special PTask system calls before and after calling underlying GPGPU runtime APIs. This allows the OS to actively monitor the state of GPU activities. The OS can exert control over EE and CE scheduling by forcing tasks to delay or sleep within the PTask system calls. GPU EEs and CEs are scheduled separately, allowing the EE and CEs of a GPU to be utilized simultaneously. PTask also supports automatic GPU allocation in multi-GPU systems. PTask includes a data-aware GPU scheduler that attempts to greedily schedule GPU computations on the “best” available GPU at the time the operation is issued, where “best” is defined by GPU capabilities (*e.g.*, speed) and data locality. However, data migration between GPUs must be performed by copying data to and from system memory (*i.e.*, PTask does not use

---

<sup>29</sup>For the sake of completeness, we note that MPS transmits data through POSIX named pipes.

peer-to-peer DMA). Rossbach *et al.* propose several fairness-based schedulers for EE and CE scheduling. The scheduling priority of a task is determined by a heuristic that considers parameters such as OS CPU scheduling priority, waiting time for GPU resources, and expected GPU operation time.

Rossbach *et al.* are somewhat radical in their approach to GPU scheduling in that they define a new programming model—GPGPU programs must be programmed to the PTask model. In contrast, Gdev achieves non-real-time GPU scheduling while remaining transparent to GPGPU program code (Kato *et al.*, 2012). Gdev replaces the manufacturer-provided device driver with its own. Gdev also offers a replacement GPGPU runtime stub library in order to prevent direct communication between tasks and GPU hardware by way of the memory-mapped interface. This stub library calls into the replacement device driver to issue GPU operations. Like the PTask framework, the OS (by way of the Gdev driver) can actively monitor the state of GPU operations and control the issuance of future ones. Gdev employs a “bandwidth-aware non-preemptive device” (BAND) scheduling algorithm to separately schedule the EE and CE. The BAND scheduler operates much like the Xen’s credit scheduler, with enhancements for managing the non-preemptive nature of the GPU engines in budget accounting. The BAND scheduler, in conjunction with unique GPU memory management features offered by Gdev, can be used to partition a physical GPU into several logical GPUs as a form of GPU virtualization. However, Gdev (as presented) does not integrate with any virtualization technologies (such as Xen), so we opt to not include it in the earlier virtualization category.

PTask and Gdev are both API-driven in that APIs, be they provided by the framework (PTask) or inserted by a stub library (Gdev), route the requests to perform GPU operations through a GPU scheduler. This incurs a scheduling overhead for every request. Menychtas *et al.* (2014) make the observation that, in cases where tasks submit all work to GPUs through a memory-mapped interface, a task can be prevented from accessing a GPU by *unmapping* the memory interface from the task’s virtual address space. A task’s attempt to issue work to a GPU can be then trapped within the OS’s page fault handler. The OS schedules the GPU by restoring the memory-mapped interface and allowing the application to return from the page fault. Menychtas *et al.* explored the implementation of several fairness-based schedulers based upon this mechanism in their GPU scheduling framework called NEON. The NEON schedulers are passive in that GPU scheduling is not in the code-path of each API call, so some scheduling overheads are avoided. Through reverse engineering of the GPU driver, Menychtas *et al.* were able to use the stock GPGPU software stack. Although admirably elegant, this approach has a drawback. Like TimeGraph, Menychtas *et al.*’s approach also operates at the



GPU command level. The scheduler does not disambiguate between commands that require an EE or CE, so it must schedule the GPU as a single processor—engines may be left idle as a result.

## **2.6 Conclusion**

Our review of the topics in this chapter should give us an appreciation for the challenges we face in realizing a multi-GPU real-time system. GPUs have unique constraints that permeate the GPGPU runtime, device driver, and GPU hardware. This marks GPU scheduling as distinct from CPU scheduling. GPUs have the potential to enable new capabilities in real-time systems if we can overcome these challenges. In this chapter, we have reviewed a variety of real-time techniques that we can use to address these issues. Namely, real-time task models, schedulability analysis, scheduling algorithms, and locking protocols. In the next chapter, we tie these techniques together to create a single cohesive and comprehensive framework for real-time GPU scheduling.

## CHAPTER 3: GPUSync<sup>1</sup>

In this chapter, we present the design of our real-time GPU scheduling framework, GPUSync. GPUSync addresses issues in the three fundamental categories of *allocation*, *budgeting*, and *integration*. Allocation issues include task-to-GPU assignment, the scheduling of GPU memory transfers, and the scheduling of GPU computations. Budgeting issues arise when tasks utilize more GPU resources than allocated. Integration issues relate to the technical challenges of integrating GPU hardware (and closed-source software) into a real-time system.

In resolving these issues, we pay careful attention to managing GPU-related parallelism. For example, modern GPUs can send data, receive data, and perform computations simultaneously. Ideally, these three operations should be allowed to overlap in time to maximize performance. Additionally, data transmissions result in increased traffic on shared buses used in parallel by other tasks. We carefully manage bus traffic to limit the effect bus contention has on other real-time tasks.

If a system has multiple GPUs, then parallelism-related issues arise when allocating GPUs. As we discussed in Chapter 1, it may be desirable to use a clustered or global GPU organization in order to avoid the utilization loss common to partitioned approaches. However, a GPU-using task may develop memory-based *affinity* for a particular GPU as it executes. In such cases, program state (data) is stored in GPU memory and is accessed by the task each time it executes on that particular GPU. This state must be *migrated* each time the task uses a GPU different from the one it used previously. Such migrations increase bus traffic and affect system-wide performance and predictability. GPUSync supports clustered and global GPU organizations, and supports efficient migration of program state between GPUs while maintaining real-time predictability.

---

<sup>1</sup> Portions of this chapter previously appeared in conference proceedings or journals. The original citations are as follows:  
Elliott, G. and Anderson, J. (2012b). Robust real-time multiprocessor interrupt handling motivated by GPUs. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 267–276;  
Elliott, G. and Anderson, J. (2013). An optimal  $k$ -exclusion real-time locking protocol motivated by multi-GPU systems. *Real-Time Systems*, 49(2):140–170;  
Elliott, G., Ward, B., and Anderson, J. (2013). GPUSync: A framework for real-time GPU management. In *Proceedings of the 34th IEEE International Real-Time Systems Symposium*, pages 33–44;  
Elliott, G. and Anderson, J. (2014). Exploring the multitude of real-time multi-GPU configurations. In *Proceedings of the 35th IEEE International Real-Time Systems Symposium*, pages 260–271.

The remainder of this chapter is organized as follows. We begin by contrasting API-driven and command-driven GPU schedulers. We then discuss eight general software architectures that may be used by an API-driven (real-time) GPU scheduler. There are tradeoffs to consider among these architectures, such as ease of implementation, robustness, and real-time correctness. We carefully weigh these tradeoffs in our selection of a general software architecture for GPUSync. Following this discussion, we describe our synchronization-based philosophy to real-time GPU scheduling. With these concepts firmly in place, we present the detailed design of GPUSync. This is done in three parts, each one addressing issues related to allocation, budgeting, and integration, respectively. We then discuss a variety of implementation challenges. We conclude with a few final remarks to summarize characteristics of GPUSync.

### **3.1 Software Architectures For GPU Schedulers**

There are two importance choices to make in the design of a GPU scheduler. We must first decide how the GPU scheduler is inserted between user applications and GPUs. We must then decide how tightly our scheduler integrates with the underlying RTOS. We make tradeoffs in implementation effort, efficiency, stability, and real-time correctness with each choice. We now explore these options and provide the rational behind the choices we make for GPUSync.

#### **3.1.1 API-Driven vs. Command-Driven GPU Schedulers**

In Chapter 2, we mentioned two general approaches to GPU scheduling. A GPU scheduler may be API-driven or command-driven. Under the API-driven model, explicit GPGPU API calls issued by user applications are routed through a GPU scheduling software layer, which decides when an API call may proceed. The scheduler is invoked by every API call that may issue work to a GPU. The GPU schedulers RGEM, GViM, gVirtuS, vCUDA, rCUDA, VOCL, MPS, PTask, and Gdev, which we discussed in Section 2.5, fall within this category. Under the command-driven model, GPGPU API calls are not scheduled. Instead, the GPU command sequences generated by API calls are scheduled. These commands must be written to an in-memory command buffer that is read by the GPU. A command-driven GPU scheduling framework controls when an application may write its buffer and/or when the buffer is read by the GPU. The GPU schedulers TimeGraph, NEON, and GPUvm, also discussed in Section 2.5, fall within this category.

We wish GPUSync to exhibit two properties:

1. The ability to separately schedule the EE and CEs of a GPU.
2. The ability to strictly arbitrate access to the EE and CEs of a GPU.

The first property enables efficient utilization of GPU resources. The second property allows us to model the GPU scheduler in real-time analysis. We may imagine both API-driven and command-driven schedulers that achieve these properties. However, these properties are far easier to achieve with an API-driven scheduler than a command-driven scheduler.

The low level GPGPU runtime APIs that issue work to a GPU are cleanly divided between those that perform memory copies (*i.e.*, CE work) and those that launch GPU kernels (*i.e.*, EE work). There is no API call that invokes the EE and CEs of a GPU in a combined sequence of operations. Under the API-driven approach, we can easily determine which engine is invoked by a given API call. This is not the case with the command-driven approach. A call to a single GPGPU API may write a sequence of GPU commands to the command buffer. After several GPGPU API calls, this buffer may contain a mix of CE and EE commands. A command-driven scheduler that separately schedules the EE and CEs of a GPU must parse the buffered commands in order to segment the buffer into EE and CE command sub-sequences. This scheduler must then separately schedule these sub-sequences on the appropriate GPU engines. Parsing the command buffer requires intimate knowledge of the structure and meaning of GPU commands. GPU manufacturers often withhold such information, so obtaining this knowledge may require a non-trivial effort to reverse engineer. Moreover, this process may have to be repeated for new versions of a given GPGPU runtime, as well as new GPU devices. Due to these challenges, it should come as no surprise that no one has yet, to the best of our knowledge, developed a command-driven GPU scheduler that separately schedules the EE and CEs of a GPU.

**GPUSync Architectural Choice #1.** The expressiveness of API-driven schedulers and the challenges associated with command-driven GPU schedulers lead us to make the following decision: *We use an API-driven GPU scheduler in GPUSync.*

### 3.1.2 Design-Space of API-Driven GPU Schedulers

In this section, we discuss several software architectures for API-driven GPU schedulers. An API-driven GPU scheduling framework must determine when an intercepted API call may proceed. This scheduling decision may be made *centrally* by a dedicated scheduling process (*e.g.*, a daemon) or *cooperatively* by the GPU-using tasks that share a GPU scheduling algorithm and scheduler state. In either case, the GPU scheduler

may be implemented in *user-space* or *kernel-space*. Also, a GPU scheduler may employ a mechanism that *enforces* GPU scheduling decisions, or it may *trust* API callers to abide by them. In summary, scheduling decisions may be: **(i)** made centrally or cooperatively; **(ii)** made in user-space or kernel-space; **(iii)** enforced or not. These choices give rise to eight general software architectures for API-driven GPU schedulers. We will describe and discuss the tradeoffs made by each approach. However, some additional background is needed before proceeding.

Common to all eight general software architectures is the use of interposed or stub libraries. An interposed library is inserted into the code paths of processes at dynamic link-time (*i.e.*, when the process is launched). The interposed library overrides the default linkage between application code the underlying GPGPU runtime.<sup>2</sup> An interposed library may invoke a GPU scheduling framework before passing an intercepted API call on to the original GPGPU runtime. A stub library is similar to an interposed library, excepting that the stub library does invoke the GPGPU runtime, but merely passes API calls onto another software component that does. Stub libraries can be employed at either static or dynamic link-time. The use of interposed and stub libraries is optional in the following architectures—an application may always implement the functionality of these libraries itself.

In our consideration of the eight general software architectures, we make two important assumptions: **(i)** the GPU device driver executes within the kernel-space of the RTOS; and **(ii)** we are not necessarily constrained by microkernel RTOS design principles. These assumptions are consistent with the technical constraints under which we prototype GPUSync (*i.e.*, a Linux-based OS with standard GPU drivers and GPGPU runtime).

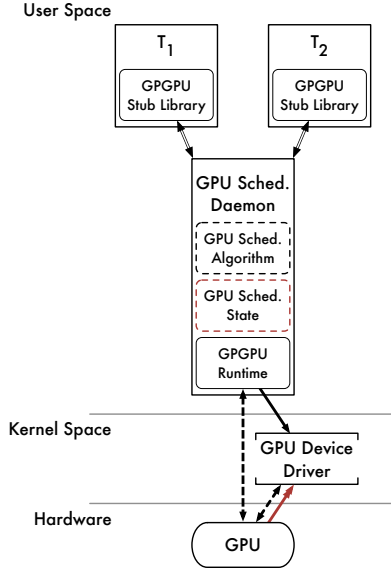
We are now ready to discuss the architectural options of GPU schedulers. Some architectures exhibit the same characteristics. In order to avoid being repetitive, we may paraphrase previously discussed characteristics. Paraphrased characteristics are denoted by square brackets (*e.g.*, “[*paraphrased characteristic*]”).

### 3.1.2.1 GPU Scheduling in User-Space

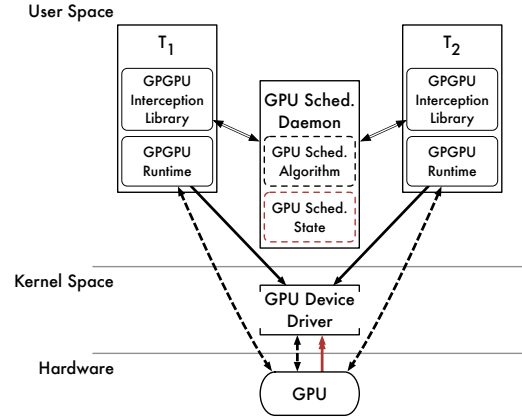
We first consider the class of software architectures for user-space GPU schedulers. Figure 3.1 depicts several such architectures. We discuss each in turn, before discussing the tradeoffs between user-space and kernel-space schedulers.

---

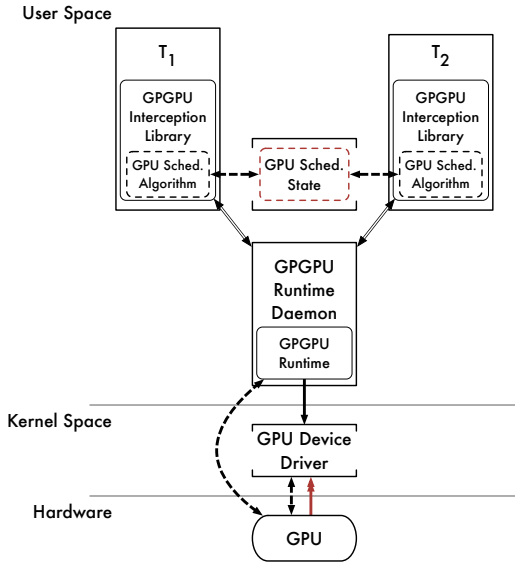
<sup>2</sup>On many UNIX-like systems, this can be accomplished through the use of the LD\_PRELOAD environment variable.



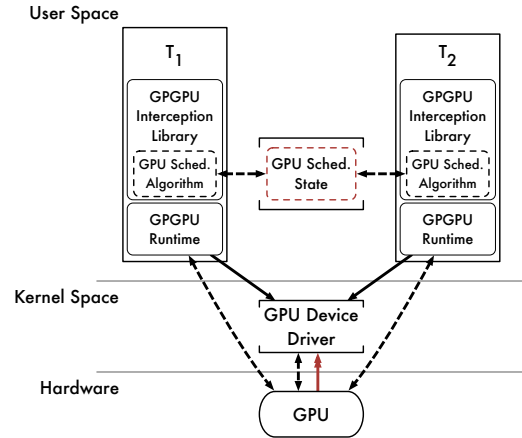
(a) Enforced centralized GPU scheduling.



(b) Centralized GPU scheduling with trusted API callers.



(c) Enforced cooperative GPU scheduling.



(d) Cooperative GPU scheduling with trusted API callers.

Figure 3.1: Several software architectures of API-driven GPU schedulers implemented in user-space.

**Centralized Scheduling With Enforcement.** Figure 3.1(a) depicts a common software architecture for GPU scheduling in user-space. Here, GPGPU API calls are issued to a GPGPU stub library within each task. The stub library redirects the API request over an IPC channel (*e.g.*, UNIX domain socket, TCP/IP socket, *etc.*) to a GPGPU scheduling daemon, which services requests according to a centralized scheduling policy. The daemon executes all API calls itself, enforcing all scheduling decisions.

This software architecture makes the following tradeoffs.

*Pros:*

1. Scheduling policy is easy to implement since decisions are centralized.
2. Scheduling decisions are enforced, since scheduled API calls are executed by the GPGPU scheduling daemon itself.

*Cons:*

1. The daemon must include, or be able to load, the GPU kernel code of constituent tasks. This can be accomplished at compile-time when the daemon is compiled. Dynamic loading of GPU kernel code is also possible, but is non-trivial to implement.
2. The IPC introduces overheads due to message passing between tasks and the daemon.
3. Barring the use of any memory remapping techniques, GPU kernel data must be transmitted through the IPC channel. Performance of data-heavy GPGPU applications (*e.g.*, a pedestrian detection application that is fed data by a video camera) is poor.
4. The daemon itself must be scheduled. This introduces additional scheduler overheads. Moreover, unless the RTOS provides a mechanism by which the daemon may inherit a priority from its constituent tasks, schedulability analysis is not straight forward. We may work around this limitation by either boosting the priority of the daemon, which hurts schedulability analysis (see Section 2.1.6.1), or we may reserve a CPU exclusively for the daemon, which results in the loss of a CPU for other work. Neither approach is desirable.

*Remarks:*

This popular architecture is employed by GViM, gVirtuS, vCUDA, rCUDA, and MPS. However, none of these implement real-time GPU scheduling policies. We note that the overheads associated with transmitting GPU kernel data over the IPC channel may lead to unacceptable performance for data-heavy applications.

**Centralized Scheduling Without Enforcement.** Figure 3.1(b) depicts another daemon-based scheduler. Here, GPGPU API calls are intercepted by an interposed library. For each call, the library issues a request for the appropriate GPU engine to the GPU scheduling daemon through an IPC channel. The library waits for each request to be granted. The daemon grants requests according to a centralized scheduling policy. The interposed library passes intercepted API calls on to the original GPGPU runtime once necessary resources have been granted.

This software architecture makes the following tradeoffs.

*Pros:*

1. Easy to implement, since scheduling decisions are centralized *and* GPU kernel code remains local to each constituent task.
2. GPU kernel data is not copied over the IPC channel. Data-heavy GPGPU applications perform well.

*Cons:*

1. GPU scheduling decisions cannot be enforced. It is possible for a misbehaved or malicious task to bypass the interposition library and access the GPGPU runtime directly.
2. [The IPC introduces message passing overheads.]
3. [The daemon itself must be scheduled.]

*Remarks:*

This architecture sacrifices the enforcement of scheduling decisions to realize performance benefits for data-heavy applications, as GPU kernel input and output data does not traverse an IPC channel. Also, this is the easiest of the eight GPU scheduling architectures to implement. This is the architecture employed by the Windows 7-based prototype of PTask, which is a non-real-time GPU scheduler. This architecture is also employed by RGEM, which is a real-time GPU scheduler.

**Cooperative Scheduling With Enforcement.** Figure 3.1(c) depicts the software architecture of a cooperative GPU scheduler and a GPGPU runtime daemon. Here, an interposed library intercepts API calls. Each instance of the library within each task invokes the same GPU scheduling algorithm, which is embedded



within the interposed library. A single instance of GPU scheduler state is stored in *shared memory*.<sup>3</sup> The interposed library passes API calls to a daemon for actual execution.

This software architecture makes the following tradeoffs.

*Pros:*

1. GPU scheduling is efficient, as each task can access the GPU scheduler state directly.
2. Scheduling decisions are weakly enforced. Although the GPGPU runtime daemon centralizes all accesses to the GPUs, misbehaved or malicious tasks may issue work directly to the daemon, bypassing the cooperative GPU scheduler.

*Cons:*

1. Access to the shared GPU scheduler state must be coordinated (or synchronized) among tasks. Depending upon the synchronization mechanism used, tasks may need to execute non-preemptively while executing scheduling algorithms (in order to avoid deadlock). This requires support from the RTOS or access to privileged CPU instructions that temporarily disable preemption.
2. A misbehaved or malicious task may corrupt GPU scheduler state, as it may overwrite any data in shared memory. Recovery from such faults may be difficult.
3. A misbehaved or malicious task may bypass the GPU scheduler and issue work directly to the GPGPU runtime daemon, unless the daemon has a mechanism by which to validate requests.
4. [The IPC introduces message passing overheads.]
5. [GPU kernel data must be transmitted through the IPC channel.]
6. [The daemon itself must be scheduled.]

*Remarks:*

Any potential benefits of cooperative scheduling are obviated by IPC-related overheads in this architecture. Moreover, the ability to enforce scheduling decisions is weakened by the fact that the GPU scheduler can be bypassed. This architecture offers *no* apparent benefit over centralized scheduling with enforcement (Figure 3.1(a)).

---

<sup>3</sup>On many UNIX-like systems, POSIX or SysV APIs may be used to allocate and manage memory shared among processes.

**Cooperative Scheduling Without Enforcement.** Figure 3.1(d) depicts the software architecture of a cooperative GPU scheduler, without the use of a daemon. Here, an interposed library intercepts API calls. As before, tasks cooperatively execute the same GPU scheduling algorithm and operate upon the same shared scheduler state. Intercepted API calls are passed on to the original GPGPU runtime when scheduled.

This software architecture makes the following tradeoffs.

*Pros:*

1. There are no IPC overheads. Data-heavy GPGPU applications perform well.
2. There is no daemon to schedule. This simplifies real-time analysis and requires less support from the RTOS.
3. [Efficient GPU scheduling.]

*Cons:*

1. [Access to GPU scheduler state must be coordinated.]
2. [GPU scheduler state is vulnerable to corruption.]
3. [GPU scheduling decisions cannot be enforced.]

*Remarks:*

This is the most efficient user-space architecture we examine. It avoids all IPC overheads. It avoids all overheads and analytical challenges raised by daemons. However, it is also the most fragile of all eight architectures. We must trust tasks to: **(i)** not bypass the GPU scheduler; and **(ii)** not corrupt the GPU scheduler state.

### 3.1.2.2 GPU Scheduling in Kernel-Space

GPU scheduling in user-space has several weakness. One weakness is that we may be unable to sufficiently protect GPU scheduler data structures. This is the case with the cooperative scheduling approaches, where GPU scheduler state can be corrupted by misbehaved or malicious tasks. However, the greatest weakness in user-space scheduling is our inability to tightly integrate with the underlying RTOS—this has the potential to prevent us from realizing a correct real-time system with any degree of confidence. RTOSs may provide some mechanisms that allow real-time tasks to affect the scheduling priority of other tasks through

user-space actions (*e.g.*, real-time locking protocols with priority-modifying progress mechanisms, as well as system calls that directly manipulate priorities). These can be leveraged to add some real-time determinism to a user-space GPU scheduler.<sup>4</sup> However, these mechanisms may be insufficient to minimize, and more importantly, *bound*, GPU-related priority inversions.

Recall from Section 2.4.5 the challenge we face in scheduling interrupt and GPGPU-runtime-callback threads. How do we dynamically bind the priorities of these threads to those of the appropriate real-time jobs, which they themselves may have priorities that change dynamically? How can we ensure that the appropriate job budgets are charged, and how do we handle budget exhaustion? Our ability resolve these issues are severely limited from user-space. These problems are best addressed by tightly integrating the GPU scheduler with the CPU scheduler and other OS components (*e.g.*, interrupt handling services) within the RTOS kernel.

These issues motivate us to consider kernel-space GPU schedulers. Figure 3.2 depicts several high-level software architectures of kernel-space GPU schedulers. We assume that all approaches benefit from the ability to tightly integrate with the RTOS. (This ability is not explicitly reflected by the diagrams in Figure 3.2.) We now discuss the tradeoffs among these kernel-space options.

**Centralized Scheduling With Enforcement.** Figure 3.2(a) depicts a software architecture with a centralized scheduler daemon. The architecture bears a strong resemblance to the one depicted in Figure 3.1(a), and functions much in the same manner. However, the GPU scheduling daemon now runs from kernel-space. This has an important implication, which we discuss shortly.

This software architecture makes the following tradeoffs.

*Pros:*

1. GPU kernel data is not copied over the IPC channel. This is possible because the GPU scheduling daemon may *directly* access the user-space memory of its constituent tasks. Data-heavy GPGPU applications perform well.
2. [Scheduling policy is centralized.]
3. [Scheduling decisions are enforced.]

*Cons:*

---

<sup>4</sup>These mechanisms may require real-time tasks to have an escalated privilege to enable priority-modifying capabilities—this is undesirable from a security perspective.

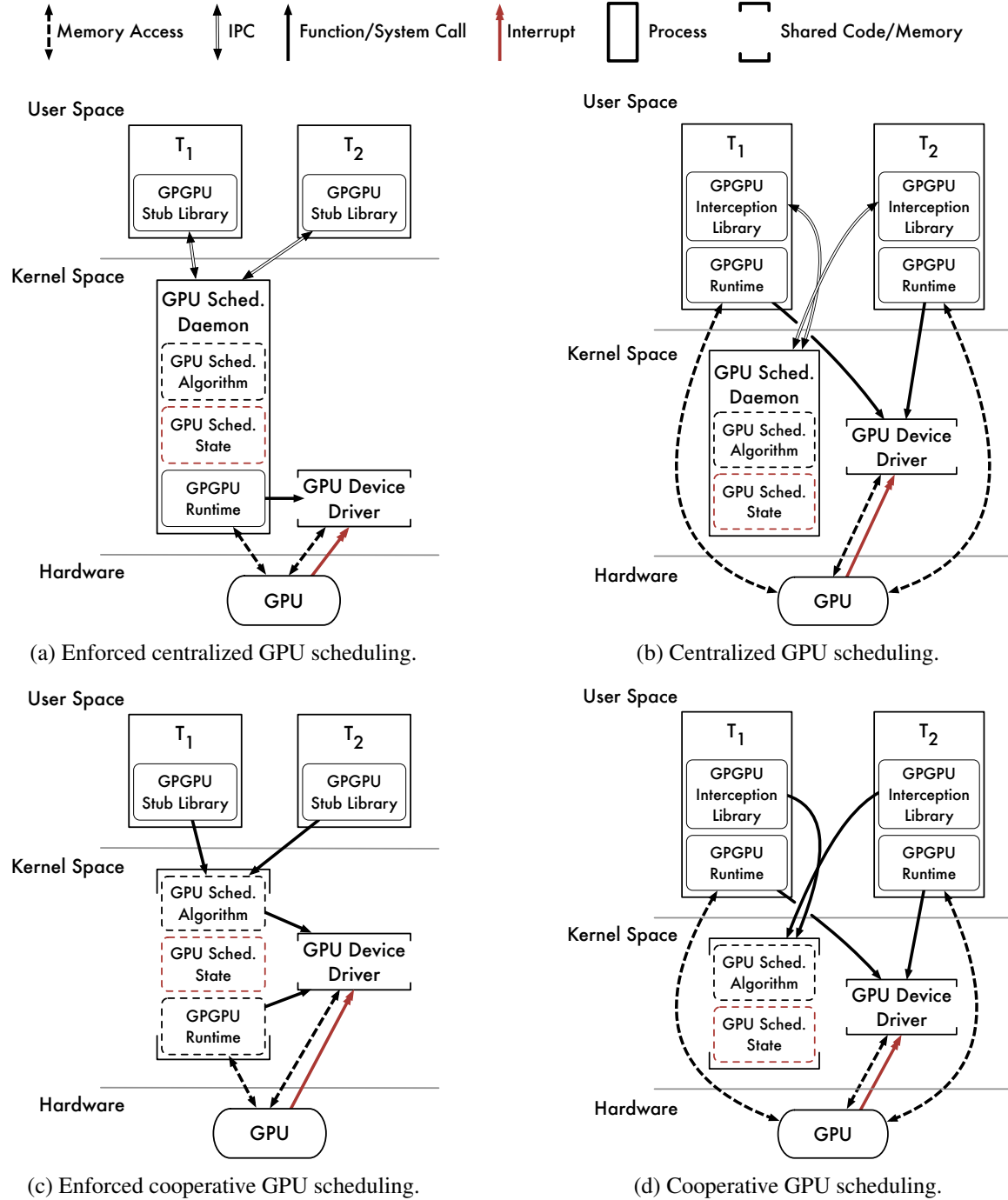


Figure 3.2: Several software architectures of API-driven GPU schedulers implemented in kernel-space.

1. A kernel-space GPGPU runtime may be unavailable. To the best of our knowledge, all manufacturer-provided GPGPU runtimes only run in user-space.
2. The daemon itself must be scheduled. This introduces additional system overheads. However, from kernel-space, we may have more flexibility in properly prioritizing the daemon to ensure real-time determinism.
3. [The daemon must include, or be able to load, the GPU kernel code of constituent tasks.]
4. [The IPC introduces message passing overheads.]

*Remarks:*

This architecture benefits from enforced centralized scheduling, without shuttling GPU kernel input and output data across the IPC channel. This approach still suffers some IPC channel overheads due to message passing. However, the greatest drawback of this approach is a practical one: the general unavailability of kernel-space GPGPU runtimes. One must develop their own—we discussed the challenges behind such an effort in Section 2.4.1. This is not an insurmountable challenge, as demonstrated by Gdev (which employs a hybrid architecture of Figure 3.2(a) and Figure 3.2(d)), but difficult.

**Centralized Scheduling Without Enforcement.** Figure 3.2(b) depicts another software architecture with a GPU scheduling daemon. Its architecture matches that of Figure 3.1(b), except that the daemon now runs in kernel-space.

This software architecture makes the following tradeoffs.

*Pros:*

1. Uses a commonly available user-space GPGPU runtime.
2. [Scheduling policy is centralized.]
3. [GPU kernel data is not copied over the IPC channel.]

*Cons:*

1. [GPU scheduling decisions cannot be enforced.]
2. [The IPC introduces message passing overheads.]

3. [The daemon itself must be scheduled.]

*Remarks:*

This architecture strikes a compromise between kernel-space centralized scheduling and practical constraints. Scheduling decisions are made within kernel-space, but carried out by individual tasks with a user-space GPGPU runtime. Thus, the architecture cannot enforce its scheduling decisions. This is the architecture employed by the Linux-based prototype of PTask, which is a non-real-time GPU scheduler.

**Cooperative Scheduling With Enforcement.** Figure 3.2(c) depicts a software architecture for a cooperative GPU scheduler. Here, GPGPU API calls are routed to a stub library that invokes the kernel-space GPU scheduler through an OS system call. GPU scheduler state is shared by all tasks, but this is stored in kernel-space data structures. Scheduled API calls are executed by a kernel-space GPGPU runtime using the program thread of the calling task.

This software architecture makes the following tradeoffs.

*Pros:*

1. GPU scheduler state is protected. Unlike cooperative user-space schedulers, the GPU scheduler state is protected within kernel-space from corruption by misbehaved or malicious user-space tasks.
2. Synchronized access to GPU scheduler state is trivial within kernel-space. There is no need for an escalated privilege to execute non-preemptively while GPU scheduler data structures are updated.
3. [GPU scheduling is efficient.]
4. [Scheduling decisions are enforced.]

*Cons:*

1. [Need for a kernel-space GPGPU runtime.]

*Remarks:*

This is the strongest of the eight architectures we examine from a performance perspective. Cooperative scheduling decisions are efficient and enforced by the RTOS. The GPGPU runtime is executed within kernel-space using the program stacks of the calling tasks, rather than a separately scheduled daemon. There are no IPC overheads. The only limitation of this approach is the reliance upon a kernel-space GPGPU runtime.

**Cooperative Scheduling Without Enforcement.** Figure 3.2(d) depicts another software architecture for a cooperative GPU scheduler. Here, an interposed library intercepts API calls, and invokes the kernel-space GPU scheduler via system calls. As before, GPU scheduler state is shared by all tasks and protected from misbehaved and malicious tasks. To schedule an API call, the GPU scheduler returns control to the interposed library. The interposed library uses the user-space GPGPU runtime to execute the scheduled API call.

This software architecture makes the following tradeoffs.

*Pros:*

1. [Uses a commonly available user-space GPGPU runtime.]
2. [GPU scheduler state is protected.]
3. [Access to GPU scheduler state is easily synchronized.]
4. [GPU scheduling is efficient.]

*Cons:*

1. [GPU scheduling decisions cannot be enforced.]

*Remarks:*

In order to support a user-space GPGPU runtime, this architecture sacrifices enforcement capabilities, trusting tasks to not bypass the interposed library by accessing the GPGPU runtime directly. Despite this limitation, it is still a strong architecture from a performance perspective. Like the prior approach, scheduling decisions are efficient. Also, there are no IPC- or daemon-related overheads. For a researcher or developer willing to implement OS-level code, this architecture is the most practical high-performance option.

**GPUSync Architectural Choice #2.** After a careful consideration of the above software architectures, we come to the following decision: *We opt to use a kernel-space cooperative GPU scheduler without enforcement in GPUSync.*

We do so because:

1. It avoids overheads due to IPCs and daemons.
2. It supports the use of user-space GPGPU runtime libraries.

3. It enables tight integration with the RTOS, enabling us to fully explore the matrix of CPU/GPU organizational choices we discussed in Chapter 1 (Figure 1.4) and support a variety of real-time schedulers.

## 3.2 Design

We now describe the design of GPUSync in detail. We begin by explaining our synchronization-based philosophy. We then describe our assumed task model supported by GPUSync. We then delve into the software design details of GPUSync.

### 3.2.1 Synchronization-Based Philosophy

GPU management is often viewed as a *scheduling* problem. This is a natural extension to conventional techniques. Similar to CPU scheduling (*e.g.*, as in Figure 2.3), pending work for GPU engines is placed in a ready queue. This pending work is prioritized and scheduled on the GPU engines. The implementation of these approaches is straightforward, but the resulting mix of CPU and GPU scheduling algorithms is difficult to analyze holistically. For example, existing schedulability analysis techniques for heterogeneous processors, such as Gai *et al.* (2002); Baruah (2004); Pellizzoni and Lipari (2007); Kim *et al.* (2013), are problematic in multi-GPU systems due to one or more of the following constraints: **(i)** they cannot account for non-preemptive GPU execution; **(ii)** they require that tasks be partitioned among different types of processors, yet our GPU-using tasks must make use of CPU, EE, and CE processors; **(iii)** they statically assign GPU-using tasks to GPUs; **(iv)** they place restrictions on how GPUs may be shared among tasks; and **(v)** they place limits on the number of CPUs and GPUs.

Instead, we view GPU scheduling as a *synchronization* problem. This perspective allows us to apply existing techniques developed for real-time locking protocols towards GPU scheduling. This perspective influences both the design of the GPU scheduler and real-time analysis. However, we wish to make clear that the distinction between scheduling and synchronization approaches is somewhat blurred: the locking protocols we use *become* GPU schedulers—these protocols prioritize ready GPU work and grant access accordingly, just as any scheduler would. The differences lie in how work is prioritized and the use of locking protocol progress mechanisms. Nonetheless, as we shall see, a synchronization-based approach gives us established techniques to address problems relating to allocation, budgeting, and integration.



### 3.2.2 System Model

We consider a system with  $m$  CPUs, partitioned into clusters of  $c$  CPUs each, and  $h$  GPUs, partitioned into clusters of  $g$  GPUs each. We assume that the workload to be supported can be modeled as a traditional sporadic real-time task system (as described in Section 2.1.1), with jobs being scheduled by a JLFP scheduler. Furthermore, we assume that the scheduled system is a SRT system for which bounded deadline tardiness is acceptable. While GPUSync’s design does not inherently preclude use in HRT systems, reliance upon closed-source software would make a claim of HRT support premature. Thus, we focus on design strategies that improve predictability and average-case performance, while maintaining SRT guarantees. We assume that tasks can tolerate GPU migration at job boundaries, and that the per-job execution times of each task remain relatively consistent, with overruns of provisioned bounds being uncommon events. We also assume that tasks pre-allocate all necessary GPU memory on any GPU upon which its jobs may run.

### 3.2.3 Resource Allocation

In this section, we discuss how GPUSync assigns GPUs and GPU engines to jobs. We begin with a high-level description of GPUSync’s resource allocation methods. We then describe the mechanisms behind GPU allocation in detail. This is followed by a description of how GPUSync arbitrates access to GPU engines.

#### 3.2.3.1 High-Level Description

GPUSync uses a two-level nested locking structure. The high-level design of GPUSync’s allocation mechanisms is illustrated in Figure 3.3. There are several components: a self-tuning execution *Cost Predictor*; a *GPU Allocator*, based upon a real-time  $k$ -exclusion locking protocol, augmented with heuristics; and a set of real-time *Engine Locks*, one per GPU engine, to arbitrate access to GPU engines.

We describe the general steps followed within GPUSync to allocate GPU resources. We refer to the schedule in Figure 3.4 to help describe when GPU resources are requested by, and allocated to, a job. This schedule corresponds to the simplified schedule for the `VectorAdd` program from Section 1.4.1 (Figure 1.3).

A *GPU critical section* is a region of code where a GPU-using task cannot tolerate a migration between GPUs during a sequence of GPU operations. A job must acquire one of  $\rho$  *tokens* associated with a particular GPU before entering a critical section that involves accessing that GPU. We call a GPU critical section

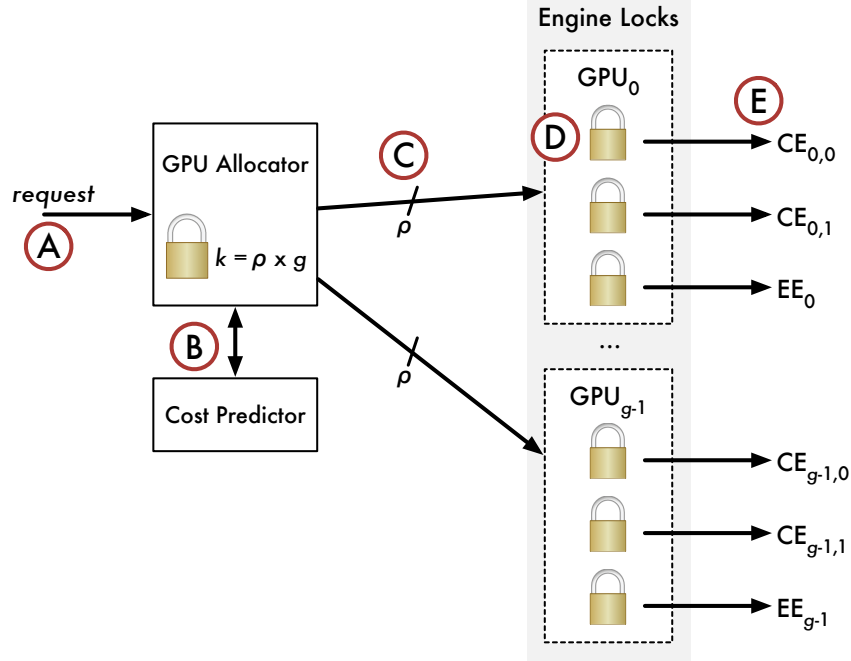


Figure 3.3: High-level design of GPUSync’s resource allocation mechanisms.

protected by a token a *token critical section*. We assume each job has at most *one* token critical section.<sup>5</sup> As depicted in Figure 3.3, a job requests a token from the GPU Allocator in Step A (or time 5 in Figure 3.4). Utilizing the cost predictor in Step B and internal heuristics, the GPU Allocator determines which token (and by extension, which GPU) should be allocated to the request. The requesting job is allowed access to the assigned GPU once it receives a token in Step C. In Step D, the job competes with other token-holding jobs for GPU engines; access is arbitrated by the engine locks. A code region protected by an engine lock is an *engine critical section*. A job may only issue GPU operations on its assigned GPU once its needed engine locks have been acquired in Step E. For example, engine locks are requested at times 11, 22, 33, and 54 in Figure 3.4. With the exception of peer-to-peer migrations, a job cannot hold more than one engine lock at a time. This is reflected in Figure 3.4, where engine locks are released at times 21, 31, 53, and 64.

The general structure of GPUSync is straightforward: a GPU Allocator assigns jobs to GPUs and engine locks arbitrate engine access. However, many questions remain. For example, how many tokens can each GPU have? What queuing structures should be used to manage token and engine requests? How can we

<sup>5</sup>Enhancements to the Cost Predictor are necessary in order to support multiple token critical sections per job—we leave such enhancements to future work.

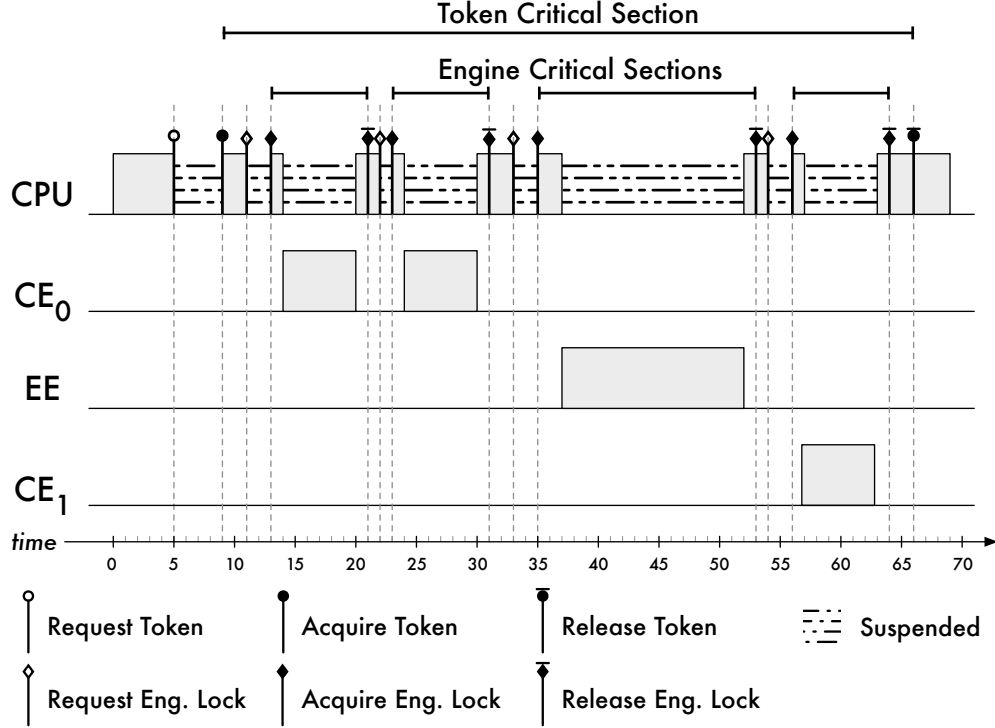


Figure 3.4: A schedule of a job using GPU resources controlled by GPUSync.

enable GPU migration, yet minimize associated overheads? We now answer such questions, and provide additional rationale for our design choices.

### 3.2.3.2 GPU Allocator

Each cluster of  $g$  GPUs is managed by one GPU Allocator; as such, we henceforth consider GPU management only within a single GPU cluster. We associate  $\rho$  tokens, a configurable parameter, with each GPU. All GPU tokens are pooled and managed by the GPU Allocator using a *single*  $k$ -exclusion lock, where  $k = \rho \cdot g$ .

The value of  $\rho$  directly affects the maximum parallelism that can be achieved by a GPU since it controls the number of jobs that may directly compete for a GPU's engines— $\rho$  must be at least the number of engines in a GPU if every engine is to ever be used in parallel. This implies that a large value should be used for  $\rho$ . However, the value of  $\rho$  also strongly affects GPU migrations. Too great a value may make the GPU Allocator too migration-averse, since tokens will likely be available for every job's preferred GPU, even those that are heavily utilized. Constraining  $\rho$  prevents GPUSync from overloading a GPU and promotes GPU migration to distribute load.

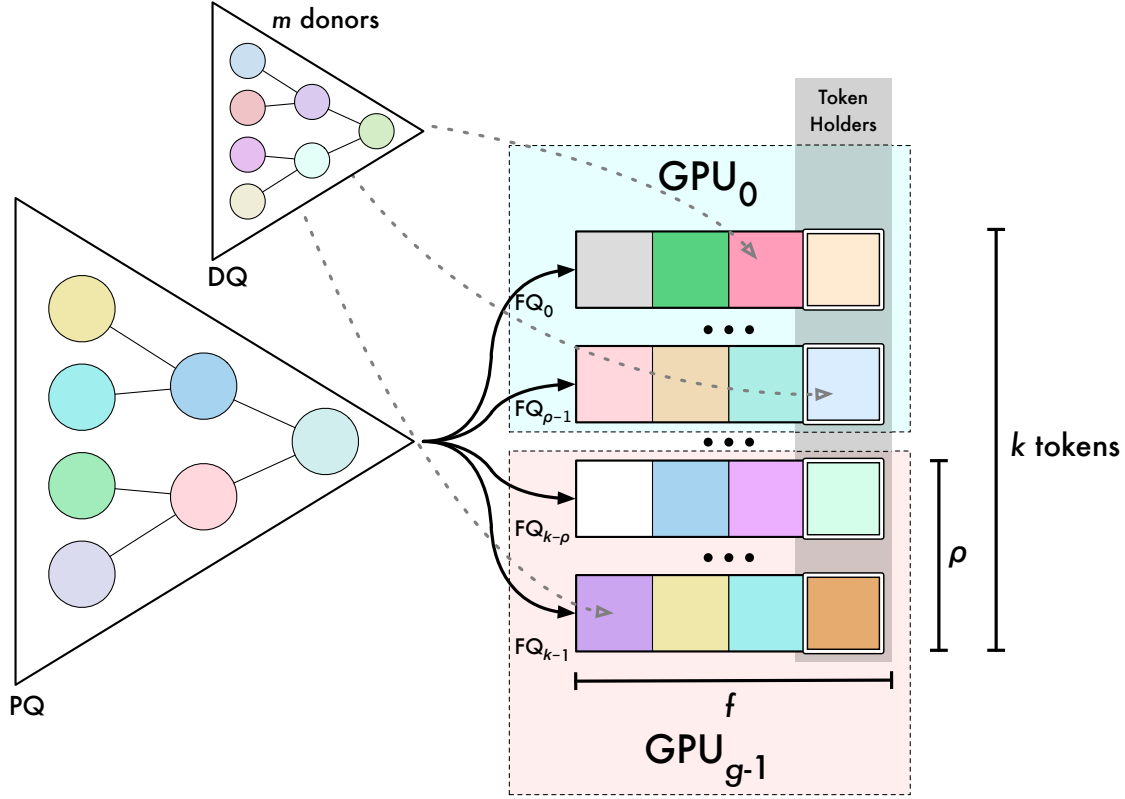


Figure 3.5: The structure of a GPU allocator lock.

The GPU Allocator’s  $k$ -exclusion lock uses a hybrid queuing structure consisting of several fixed-size FIFO queues  $FQ_i$ , a priority queue PQ, and a donor queue DQ, as depicted in Figure 3.5. The FIFO queue  $FQ_i$  is associated with the  $i^{\text{th}}$  token. Token-requesting jobs are enqueued in a load-balancing manner until every FQ is  $f$  jobs in length. (We will discuss mechanisms for load balancing shortly.) The parameter  $f$  is configurable. Additional requests that cannot be placed in an FQ “overflow” into PQ or DQ. Jobs are moved from PQ or DQ into an FQ as space becomes available. A job enqueued in  $FQ_i$  suspends until it is at the head of  $FQ_i$ , in which case it is granted the  $i^{\text{th}}$  token.

The GPU Allocator adopts the structure and rules of the R<sup>2</sup>DGLP (Section 2.1.7.2), with some exceptions. There are three key differences. First, unlike the R<sup>2</sup>DGLP, the maximum length of the GPU Allocator FIFO queues is configurable by the system designer. Selecting different values for  $f$  allows the GPU Allocator lock to take on different analytical properties. The GPU Allocator functions exactly as the R<sup>2</sup>DGLP when  $f = \lceil c/k \rceil$ , as the  $k$ -FMLP when  $f = \infty$ , and as a purely priority-based protocol when  $f = 1$ . A system

designer may tailor the GPU Allocator to their specific task sets and schedulers.<sup>6</sup> For instance, the k-FMLP can outperform the R<sup>2</sup>DGLP in some cases (see Ward *et al.* (2012) for more information). The purely priority-based GPU Allocator may be used with a task-level static priority scheduler, such as RM, when request interference on high-priority tasks must be minimized.

The second difference between the GPU Allocator and the R<sup>2</sup>DGLP is that, for cases where a GPU cluster is shared by tasks of different CPU clusters, we adopt Rule C1 (*i.e.*, the donation-at-job-release rule) of the CK-OMLP. In this way, the GPU Allocator can be configured to mimic the CK-OMLP by using  $f = \lceil (c \cdot \gamma) / k \rceil$ , where  $\gamma$  is the number of CPU clusters that share the GPU cluster in question. Such a configuration differs from the CK-OMLP in that the GPU Allocator uses  $k$  shorter FIFO queues instead of a single long one—this change does not alter bounds on pi-blocking. Also, the “request stealing” aspect of the R<sup>2</sup>DGLP’s Rule R5 ensures similar work-conserving behavior at runtime. The addition of Rule C1 subsumes the inheritance and donation rules of the R<sup>2</sup>DGLP. This is because Rule C1 ensures that token-holding jobs are always scheduled on a CPU when ready to run—there is no need for other progress mechanisms. We note that valid values of  $f$  must meet the constraint  $f \geq \lceil m/k \rceil$  when Rule C1 is enacted.

The last difference between the GPU Allocator and the R<sup>2</sup>DGLP is that the GPU Allocator augments Rules R1a, R1c, and R5 of the R<sup>2</sup>DGLP with heuristics to improve runtime performance. We discuss the details of these heuristics next.

Real-time locking protocols are rarely designed to make use of online knowledge of critical section lengths, even though critical sections figure prominently in schedulability analysis and system provisioning. We augment the GPU Allocator lock to incorporate knowledge of token critical section lengths into queuing decisions to reduce the frequency and cost of GPU migrations without preventing beneficial migrations. Most of our heuristics use information provided by the Cost Predictor. For now, let us treat the Cost Predictor as a black-box that returns a predicted token critical section length for a GPU request  $R_i$  by a job  $J_i$ , under the hypothesis that  $R_i$  is enqueued on  $FQ_x$  and is eventually allocated the associated token—this prediction includes the cost of migration. We say that the *preferred GPU* of a job  $J_i$  is the GPU last used by task  $T_i$ , since  $J_i$  may have affinity with this GPU. Let the function  $perf(J_i)$  return the identity of the GPU preferred by  $J_i$ .

We must define several additional terms and functions in order to describe our heuristics. Let  $L_i$  denote the provisioned token critical section length for  $R_i$ . Let  $L_{i,x}^{prd}$  denote the *predicted* token critical section

---

<sup>6</sup>We caution that not every configuration of the GPU Allocator (and GPUSync, in general) is necessarily amenable to real-time analysis.

length provided by Cost Predictor, supposing  $R_i$  is enqueued on  $FQ_x$ . We assume  $0 \leq L_{i,x}^{prd} \leq L_i$ . We use two methods to measure the length of  $FQ_x$ . The first method uses the same function as the R<sup>2</sup>DGLP (Equation 2.33), measuring length in terms of the number of enqueued requests. We repeat it here for a consolidated presentation:

$$length(FQ_x) \triangleq |FQ_x|. \quad (3.1)$$

The second method measures the length in terms of the token critical section length predictions provided by the Cost Predictor:

$$execLength(FQ_x) \triangleq \sum_{R_j \in FQ_x} L_{j,x}^{prd}, \quad (3.2)$$

where we reindex the requests in  $FQ_x$  with  $j$ . For a given GPU, we determine the set of tasks that make up its “active users.” The set of tasks that last used, or are currently using, a GPU make up the GPU’s active users. A task with no pending job can still be among the active users of a GPU because the *next* job of this task is assumed to prefer the GPU used by the task’s prior job. Let the function  $numGpuUsers(FQ_x)$  denote the number of active users of the GPU associated with  $FQ_x$ . We denote the distance between two GPUs with function  $distance(GPU_a, GPU_b)$ .<sup>7</sup> Finally, let the set  $\mathcal{F} \triangleq \{FQ_x \mid length(FQ_x) < f\}$ , denote the set of FQs that are not full at any given time instant. The GPU Allocator employs the following four heuristics. We use these heuristics simultaneously, as each controls a different operation within the GPU Allocator.

### H1: Distribute Tasks Among GPUs

Conditions: This heuristic is employed when: **(i)** the GPU token request  $R_i$  is the first such request of any job of task  $T_i$ ; and **(ii)** there exist multiple FQs that are not full (i.e.,  $|\mathcal{F}| \geq 2$ ).

Description: Request  $R_i$  is enqueued on an FQ from the set  $\min_{FQ_x \in \mathcal{F}} \{numGpuUsers(FQ_x)\}$ . That is, the request is enqueued on an FQ of a GPU with the fewest number of current users. There may be several FQs that satisfy this criteria. The particular FQ is selected using the same methods of Heuristic H2.

Rational: GPUSync is designed with sporadic task sets in mind. If a GPU-using task does not currently require a GPU, there remains a high likelihood that it will in the near future. This heuristic exploits this knowledge to distribute tasks among GPUs to reduce contention for preferred GPUs.

---

<sup>7</sup>Recall from Section 2.4.2.2 that distance is the number of PCIe links to the nearest common switch or I/O hub of two GPUs.

## H2: Minimize Predicted Response Time

Conditions: This heuristic is employed when: (i) job  $J_i$  issues request  $R_i$  for a GPU token; and (ii) there exist multiple FQs that are not full.

Description: Request  $R_i$  is enqueued on an arbitrary FQ from the set  $\min_{FQ_x \in \mathcal{F}} \{execLength(FQ_x) + L_{i,x}^{prd}\}$ .

Rational: This heuristic is meant to only allow a GPU migration if it is predicted to be *beneficial*. A GPU migration is only beneficial if it expedites the *completion* of the token critical section of job  $J_i$ . Harmful GPU migrations are possible if we only expedite token acquisition. This heuristic not only estimates how long  $J_i$  must wait for a token, given by  $execLength(FQ_x)$ , but also how long it may take for  $J_i$  to complete its token critical section, given by  $L_{i,x}^{prd}$ , which includes predicted migration costs.

## H3: Affinity-Aware Priority Donation

Conditions: This heuristic is employed when: (i) job  $J_i$  issues request  $R_i$  for a GPU token; (ii) all FQs are full (*i.e.*,  $|\mathcal{F}| = k$ ); and (iii) job  $J_i$  must become a priority donor.

Description: We denote the set of eligible donee jobs by  $\mathcal{E}$ . Let eligibility be defined by Rule R1c of the R<sup>2</sup>DGLP. All eligible donees have requests in the FQs. Let  $x$  denote the index of the FQ in which an eligible donee job  $J_j$  has a request (*i.e.*,  $FQ_x$ ). We select a donee from the set generated by  $\min_{J_j \in \mathcal{E}} \{distance(perf(J_i), GPU_{\lfloor x/g \rfloor})\}$ .<sup>8</sup> That is, we select an eligible donee with a request in an FQ of a GPU that is closest to the preferred GPU of  $J_i$ . If there are multiple such eligible donees, then we select the one with the earliest predicted token critical section completion time.

Rational: Rule R1c dictates that a donor is immediately moved to an FQ when its donee completes its critical section. This heuristic is meant to increase the likelihood that the request  $R_i$  is moved from the DQ to an FQ of job  $J_i$ 's preferred GPU, or failing this, one nearby.

Additional Remarks: This heuristic is described in terms of priority donation as defined by the R<sup>2</sup>DGLP. However, we may modify this heuristic to apply to configurations where the GPU Allocator is accessed by jobs on different CPU clusters, *i.e.*, when Rule C1 of the CK-OMLP is used, subsuming Rule R1c. In such a configuration, this heuristic may be employed when a released job  $J_i$  has a preferred GPU and  $J_i$  must become a priority donor, as defined by Rule C1. We note, however, that eligible donees may include jobs that hold other inter-CPU-cluster (non-GPU-token) shared resources. This heuristic

---

<sup>8</sup>We compute the index of a GPU in a cluster from the index of an FQ by dividing the FQ-index by the number of GPUs managed by the GPU Allocator and rounding down to the nearest integer.

may be updated to avoid donating a priority to such jobs whenever possible. We also note that there is a delay between when  $J_i$  begins execution (after it is no longer a priority donor) and when it issues its GPU token request  $R_i$ —the FQ of  $J_i$ 's donee may have been filled by other requests by the time  $R_i$  is issued. Ultimately, Heuristic H3 is very difficult to implement under Rule C1. Practically speaking, it may be best to do without it in this case, unless there is a clear demonstrated need.

#### **H4: Affinity-Aware Request Stealing**

Conditions: This heuristic is employed when: (i) an  $FQ_x$  is empty after the token for  $FQ_x$  has been released; (ii) the PQ and DQ are empty; and (iii) there exists an  $FQ_y$  with  $length(FQ_y) > 1$ .

Description: Let  $w_j$  denote the predicted time from completion if request  $R_j$  if it remains in its current FQ. This prediction is computed by summing the predicted token critical section lengths of requests enqueued ahead of  $R_j$  and  $R_j$  itself. Let the function  $fq(R_j)$  return the index of the FQ in which  $R_j$  is enqueued. Let  $\mathcal{W}$  denote the set of unsatisfied requests of all FQs, *i.e.*, all requests waiting to become a resource holder. Let  $\hat{\mathcal{W}}$  denote the set of requests that are predicted to benefit by obtaining the token of  $FQ_x$  immediately, relative to waiting for a token in their respective FQs. Thus,  $\hat{\mathcal{W}} \triangleq \mathcal{W} \setminus \{R_j \mid w_j > L_{j,x}^{prd}\}$ . An arbitrary request  $R_z$  from the set  $\max_{R_j \in \hat{\mathcal{W}}} \{L_{j,fq(R_j)}^{prd} - L_{j,x}^{prd}\}$  is removed from its associated FQ, say  $FQ_y$ , and moved to  $FQ_x$ . If the resource holder of  $FQ_y$  inherited the priority of  $R_z$ , then this inheritance relationship is ended, and the resource holder may inherit a new priority from another request in  $FQ_y$ . It is possible for  $\hat{\mathcal{W}} = \emptyset$ , in which case no request is moved to  $FQ_x$ .

Rational: This heuristic is meant to immediately grant the available token of  $FQ_x$  to an unsatisfied request in another FQ. The heuristic selects the request that is expected to benefit the most. In cases where migration costs are high, it may be better to simply let  $FQ_x$  idle, which this heuristic allows.

We briefly summarize the above heuristics. Heuristic H1 assigns the initial GPU to tasks in order to distribute GPU affinity among the GPUs. Heuristic H2 enqueues request  $R_i$  in the FQ that the Cost Predictor predicts will lead to the earliest completion time of  $R_i$ . Heuristic H3 guides priority donation towards jobs that have (or are slated to obtain) a GPU that minimizes migration costs for request  $R_i$ , increasing the likelihood that  $R_i$  will obtain the same GPU. Finally, Heuristic H4 grants a newly idle token to a request  $R_i$  that is waiting for another token, but only if doing so is expected to expedite the completion of  $R_i$ .



### 3.2.3.3 Cost Predictor

The Cost Predictor provides predictions of the token critical section lengths of GPU-using jobs. These predictions include the estimated cost of GPU migration. The Cost Predictor uses a record of observed token critical section lengths to make its predictions.

The Cost Predictor makes estimates of token critical section lengths by tracking the accumulate CPU *and* GPU execution time within the token critical section. Upon token assignment, the GPU Allocator informs the Cost Predictor when it has allocated a GPU token to a job  $J_i$ , along with the identity of the assigned GPU. In response, the Cost Predictor resets an execution-time counter, records the identity of the newly assigned GPU, and notes the distance between the last GPU used by task  $T_i$  and the newly assigned GPU. The distance information enables the Cost Predictor to make different predictions for GPU migrations of different distances.

The execution-time counter tracks the combined CPU and GPU execution time of a job while a token is held. Execution delays due to preemption and blocking due to engine lock acquisition (explained later) are *not* included, but CPU suspension durations due to GPU operations (memory copies, GPU kernels) *are*. This tracking requires tight integration with the CPU scheduler.

When a job releases its GPU token, the accumulated execution time gives a total request execution cost for both CPU and GPU operations and includes any delays due to migrations. However, this measurement is for only a single observation and provides a poor basis for predicting future behavior, especially since these measurements are strongly affected by PCIe bus congestion. Thus, we use a more refined process to drive a prediction model based upon statistical process control (Kim and Noble, 2001). Specifically, for each task and GPU migration distance pair, we maintain an average and standard deviation over a window of recent observations (our experimentation suggests that a window of twenty samples is suitable). The average and standard deviation is recomputed after every new observation unless the observation falls more than two standard deviations away from the average and at least ten prior observations have been made. This filtering prevents unusual observations from overly-influencing future predictions.

One may be concerned that the heuristic nature of GPUSync’s Cost Predictor does not lend itself to real-time predictability. However, the Cost Predictor derives its predictions from an *average* of observed token critical section lengths. Assuming tasks never exceed provisioned execution times, the Cost Predictor can never make a prediction that exceeds the provisioned execution time. Under-estimates are possible. The

effects of under-estimates are already accounted for in pi-blocking bounds derived from worst-case blocking analysis.

#### 3.2.3.4 Engine Locks

Engine locks enable the parallelism offered by GPUs to be exploited while obviating the need for the (unpredictable) GPU driver or GPU hardware to make resource arbitration decisions.

A mutex is associated with each GPU copy and execution engine, as depicted in Figure 3.3. For GPUs with *two* copy engines, one engine lock must be obtained before copying data *to* the GPU, and the other must be obtained before copying data *from* the GPU. We require that all issued GPU operations are *completed* before the associated engine lock is released. This is necessary in order to: **(i)** prevent the GPU driver or hardware from interfering with GPUSync’s scheduling decisions; and **(ii)** support proper scheduling of interrupt daemons and GPGPU callback threads (we discuss this shortly in Section 3.2.5). For GPUs that do not ensure engine independence (Section 2.4.4), we also require that application code that issues several GPU operations to the same engine within the same engine critical section wait for each operation to complete before the next is issued. This is necessary to prevent the GPU hardware scheduler from stalling the other engines.

Engine locks should be held for as little time as possible in order to prevent excessive blocking times that degrade overall schedulability. For this reason, tasks are discouraged from issuing multiple GPU operations within the same engine critical section, although this is not strictly prevented by GPUSync. Minimizing the hold time of execution engine locks requires application-specific solutions to break kernels into small operations. However, a generic approach is possible for copy engine locks. *Chunking* is a technique where large memory copies are broken up into smaller copies. The effectiveness of this technique is demonstrated by Kato *et al.* (2011a). GPUSync supports chunking of both regular and peer-to-peer memory copies by way of a user-space library. Chunk size is fully configurable. The library also transparently handles copy engine locking.

GPUSync can be configured to satisfy engine lock requests in either FIFO- or priority-order (all GPUs within the same GPU cluster must use the same engine lock order). Blocked jobs suspend while waiting for an engine. A job that holds an engine lock may inherit the *effective* priority of any job it blocks. We stress effective priority, because the blocked job may itself inherit a priority from a job waiting for a token. In

order to reduce worst-case blocking, a job is allowed to hold at most *one* engine lock at a time, *except* during peer-to-peer migrations.

**Bus Scheduling Concerns.** The copy engine locks indirectly impose a schedule on the PCIe bus if we assume that bus traffic from non-GPU devices is negligible. This is because these locks grant permission to send/receive data to/from a particular GPU to one task at a time. It is true that traffic of other GPUs will cause bus congestion and slow down a single memory transmission. However, since the PCIe bus is packetized, it is shared fluidly among copy engine lock holders. Thus, we can bound the effect of maximum PCIe bus congestion and incorporate it into real-time analysis. Deeper analysis or explicit bus scheduling techniques may be necessary if very strict timing guarantees are required, but such approaches are beyond the scope of this dissertation.

**Migrations.** GPUSync supports both peer-to-peer and system memory migrations, which are handled differently.

For a peer-to-peer migration from one GPU to another, a job must hold copy engine locks for both GPUs. Requests for both copy engine locks are issued together *atomically* to avoid deadlock.<sup>9</sup> This is accomplished through the use of dynamic group locks (DGLs) (Section 2.1.6.2). The job may issue memory copies to carry out the migration once both engine locks are held. Peer-to-peer migration isolates traffic to the PCIe bus: copied data does not traverse the high-speed processor interconnect or system memory buses—computations utilizing these interconnects are not disturbed. However, gains from fast peer-to-peer migrations may be offset by higher lock contention: unlikely scenarios exist where every token holder in a GPU cluster may request the same copy engine lock simultaneously. However, unlikely scenarios must still be accounted for in schedulability analysis.

If GPUSync is configured to perform migrations through system memory, then such migrations are performed *conservatively*, *i.e.*, they are always assumed to be necessary. Thus, state data is aggregated with input and output data. State is always copied off of a GPU after per-job GPU computations have completed. State is then copied back to the next GPU used by the corresponding task for its subsequent job if a different GPU is allocated. An advantage of this approach over peer-to-peer migration is that a job never needs to hold two copy engine locks at once. This conservative approach may seem heavy handed, especially when migrations between GPUs may not always be necessary. An optimistic approach could be taken where state

---

<sup>9</sup>GPUSync also *grants* requests atomically to avoid deadlock if priority-ordered engine locks are used.

is only pulled off of a GPU once a migration is detected. However, this results in the same degree of copy engine lock contention as peer-to-peer migrations, but without the benefits of isolated bus traffic.

**Critical Section Fusion.** Every GPU DMA memory copy must be protected by a copy engine critical section. For chunked memory copies, a call to unlock a held copy engine lock may be immediately followed by a call to reacquire the very same lock. There are cases where this results in needless system call overheads, or worse, harm schedulability. For FIFO-ordered engine locks, such a case occurs when the engine lock in question is not contended. For priority-ordered engine locks, this occurs when said engine lock has been requested by only tasks with lower priorities than the engine lock holder.

In order to avoid needless overheads and improve schedulability, GPUSync provides a system call interface that allows an engine-lock-holding callee to ask the question “Should I relinquish this lock?” (or in the case of DGLs, “Should I relinquish these locks?”). Each engine lock instance answers this question in accordance with its locking protocol (*e.g.*, a FIFO-ordered engine lock always answers “yes” if the lock is contended). If the answer is “yes,” then the callee should issue the necessary call to release its held engine lock(s).<sup>10</sup> Otherwise, the callee may proceed *directly* into its next critical section without releasing and reacquiring its needed engine lock(s). This essentially *fuses* back-to-back engine lock critical sections of a job that are protected by the same locks. Real-time correctness is maintained by the “preemption points” inserted between each original critical section.

Critical section fusion for FIFO-ordered engine locks is primarily a runtime optimization. However, fusion is important for priority-ordered engine locks, since it can strongly impact schedulability. Consider the case where two tasks with different priorities contend for the same copy engine lock and both need to perform a large chunked memory copy. Without fusion, the high and low priority chunks are interleaved by the copy engine lock. This is due to that the low priority task may obtain the contended copy engine lock after each time the high priority task exits a copy engine critical section. The locking protocol governing the engine lock cannot know that the high priority task intends to immediately re-request the lock, since this behavior is entirely application-defined. As a result, each copy engine lock request issued by the high priority task may be delayed by one low priority copy engine critical section. The high priority task may suffer a priority inversion each time it requests the copy engine. However, with fusion, the high priority task may

---

<sup>10</sup>We say “should” because the GPUSync cannot strictly enforce GPU scheduling decisions, we as discussed at the end of Section 3.1.2.2.

suffer a priority inversion for only its first copy engine request, since it retains ownership of the engine lock until its memory copies have completed. This can be reflected in schedulability analysis.

### 3.2.4 Budget Enforcement

Budget enforcement policies are necessary to ensure that a task’s resource utilization remains within its provisioned budget. These policies are particularly important in a real-time GPU system that uses closed-source GPU drivers and GPGPU runtimes. We first explain why execution time variance is usually due to closed-source software, rather than GPU kernels or GPU hardware. We then describe the budget enforcement policies provided by GPUSync.

GPU kernels usually exhibit consistent runtime behaviors. There are two reasons for this. First, high-performance GPU algorithms usually depend upon very regular, non-divergent, execution patterns (recall our discussion of the GPU EE in Section 2.4.2.1). For example, image processing algorithms usually fall within this category, since computation commonly consists of applying the same operation to image pixels. The second reason for GPU kernel execution time predictability stems from the relative simplicity of the EE. Deep execution pipelines, branch predictors, memory prefetchers, and multiprocessor cache interference all contribute towards variance in execution time on modern multiprocessor CPUs. However, in order to maximize the number of transistors devoted to computation, the SMs of the EE eschew such average-case-oriented features. For example, all CUDA-capable NVIDIA GPUs to date lack L1 cache coherency among SMs.

GPU DMA operations also exhibit relatively consistent runtime behaviors. The execution time of DMA operations *are* affected bus contention. However, as we show in Section ??, it is possible to perform experiments to characterize worst-case behavior. Such experimental results may be incorporated into schedulability analysis and budget provisioning. We cannot apply the same approach to program code, in general.

In most cases, we can rule out GPU kernels or GPU hardware as the primary cause of overruns of provisioned budgets. Ignoring user application software, this leaves the closed-source software as the most likely culprit. This is understandable since this software has not been designed with real-time constraints in mind—its implementation optimizes for average-case performance, not worst-case performance. As a result, the execution time of operations that typically exhibit predictable runtime behavior may occasionally take much longer to execute than usual. For example, we show that this is the case for the interrupt-handling

subsystem of a GPU driver in Section ???. We have already discussed the strong practical motivations for using closed-source GPU software in a real-time system. However, we must recognize that the closed-source software may sometimes act unpredictably. We must prepare for it.

Understanding the likely causes of budget overruns in a real-time GPU system can make us better researchers and engineers, but it is not clear how this information can guide the design of budget enforcement policies in GPUSync. Ultimately, it matters little if a budget overrun is due to user code, closed-source software, GPU kernels, or GPU hardware. The consequence is the same: a budget is overrun. Thus, GPUSync uses general budget enforcement policies that work regardless of whether an overrun is due to CPU or GPU work.

Unfortunately, strict budget enforcement is difficult, if not impossible, to achieve due to limitations of the GPU technology. Non-preemptivity of GPU operations makes budget enforcement problematic. Even if limited preemption is provided by breaking a single GPU operation into multiple smaller operations, data on a GPU may be in a transient state at a preemption point and thus be too difficult (or too costly) to resume on another GPU at a later time. This motivates us to focus on budget enforcement based on overrun *recovery* rather than strict enforcement to absolutely prevents overruns.

GPUSync provides three budget enforcement options: signaled overruns, early budget releasing, and a bandwidth inheritance-based method.

**Signaled Overruns.** Under the signaled overrun policy, jobs are provisioned with a single budget equal to a maximum CPU execution time, plus a maximum total GPU operation time. A job's budget is drained when it is scheduled on a CPU or GPU engine. The OS delivers a signal to the job if it exhausts its budget. In order always maintain a consistent state with the GPGPU runtime, this signal is deferred if the job holds an engine lock, and it is delivered immediately once the lock is released. The signal triggers the job to execute an application-defined signal handler. The handler is application-defined since appropriate responses may vary at different points of execution. As depicted in Figure 3.6, one way an application may respond to a signal is to unwind the job's stack (either by throwing an exception<sup>11</sup> or a call to the standard function `longjmp()`) and execute clean-up code before releasing its token lock.

---

<sup>11</sup>Throwing exceptions from signal handlers may require platform support and special compiler options. For example, one must specify the compiler options `-fasynchronous-unwind-tables` and `-fnon-call-exceptions` to the GCC compiler.

```

▷ Enabled/disabled on try-block enter/exit.
function BUDGETSIGNALHANDLER()
    throw BudgetException();
end function

procedure DOJOB()
     $t \leftarrow \text{GetToken}()$ ;
     $gpu \leftarrow \text{MapTokenToGpu}(t)$ ;
    try:
        DoGpuWork( $gpu$ );           ▷ Main job computation.
    catch BudgetException:
        CleanUpGpu( $gpu$ );         ▷ Gracefully cleans up state.
    finally:
        FreeToken( $t$ );
end procedure

```

Figure 3.6: Example of budget signal handling.

**Early Releasing.** The “early releasing” policy immediately refreshes the exhausted budget of a job with the budget of the task’s *next* job. That is, the budget of the next job is released early. This is accomplished by shifting (postponing) the job’s current deadline to that of its next job. In essence, the next job has been sacrificed in order to complete the overrunning one. This policy penalizes an overrunning task by forcing it to consume its own future budget allocations. This prevents the system from being overutilized in the long-term. Under deadline schedulers, deadline postponement also helps to prevent the system from being overutilized in the short-term. We note that deadline postponement is challenging to implement since it requires priority inheritance relations established by locking protocols to be reevaluated.

**Bandwidth Inheritance (BWI).** A job that overruns its budget while holding a shared resource can negatively affect other jobs, even non-resource-using ones. We use BWI to limit the effects of such overruns to resource-sharing tasks. This is accomplished by draining the budget of a *blocked* job whose priority is currently inherited by a *scheduled* job in place of the scheduled job’s own budget.<sup>12</sup> This can cause a blocked job to be penalized for the overrun of another, but improves temporal isolation for non-GPU-using tasks. Under deadline-based scheduling, GPUSync takes additional measures to isolate the temporal effects of overruns—specifically, the “abort/refresh/reissue” BWI technique we discussed in Section 2.1.6.1. If the budget of a job waiting for a GPU token is exhausted, then: (i) the token request is immediately aborted; (ii) the budget of the exhausted job is refreshed through early releasing, decreasing the job’s priority; and

---

<sup>12</sup>Jobs must be provisioned with additional budget derived from analytical bounds on pi-blocking to cover budget that may be lost via BWI under assumed conditions.

(iii) the token request is automatically reissued. Care must be taken in the implementation of this policy, as decreasing the priority of the exhausted job also requires priority inheritance relations established by locking protocols to be reevaluated.

### 3.2.5 Integration

Resource allocation techniques and budget enforcement policies make up the general elements of GPUSync that may be applied to a variety of GPU technologies (*i.e.*, GPUs from different manufacturers and GPGPU runtimes) and RTOSs (when RTOS code is available). We now discuss elements of GPUSync that address issues that arise due to reliance on specific non-real-time software. We discussed the deficiencies of split interrupt handling in the Linux kernel in Section 2.2.3.1 and the challenges posed by the CUDA runtime’s callback threads in Sections 2.4.1 and 2.4.5. In this section, we discuss how GPUSync resolves these issues to improve real-time predictability. We present first the method we use to realize proper real-time scheduling of GPU interrupts. We then apply similar techniques to schedule CUDA callback threads.

#### 3.2.5.1 GPU Interrupt Handling

We must define scheduling policies for the execution of both top- and bottom-halves of GPU interrupts. We discuss these in turn.

**GPU Top-Halves.** The disruptions imposed upon a real-time system by interrupt top-halves are inescapable. However, we can isolate their effects in order to improve real-time predicability. We do so through *CPU shielding*. CPU shielding allows the system designer to direct interrupts to a particular CPU, or group of CPUs. To support this, the OS associates a CPU bitmask with every interrupt source, or more generally, each interrupt identifier (*i.e.*, a unique interrupt ID). These bitmasks are used by the OS to program the underlying interrupt-handling hardware (*e.g.*, the Advanced Programmable Interrupt Controller (APIC)) that is responsible for delivering interrupts to each CPU.

Under GPUSync, we assume that the system designer applies the appropriate CPU bitmasks to direct interrupts of a given GPU to the CPUs that may be scheduled with tasks that use that GPU. For example, suppose GPU<sub>*i*</sub> is used by tasks that are scheduled on CPU<sub>*j*</sub> and CPU<sub>*k*</sub>. The system designer would configure the CPU bitmasks to ensure that the interrupts of GPU<sub>*i*</sub> may only be handed by CPU<sub>*j*</sub> and CPU<sub>*k*</sub>. CPU<sub>*j*</sub> and CPU<sub>*k*</sub> do not necessarily reside in the same CPU cluster. For instance, GPU<sub>*i*</sub> could be shared by CPU<sub>*j*</sub> and



$CPU_k$  under partitioned CPU scheduling. In this case,  $CPU_j$  could be forced to handle GPU interrupts on behalf of tasks that execute exclusively on  $CPU_k$ —this must be accounted for in schedulability analysis.

Under older interrupt handling mechanisms, a GPU may be forced to share the same interrupt identifier with other GPUs and even other devices. This stems from limitations in legacy hardware where each interrupt identifier maps to a physical interrupt pin or wire. This leads to scenarios where a GPU may be forced to share an interrupt identifier with unrelated devices, such as network interface cards and disk controllers. This sharing may make it difficult to isolate GPU interrupts on the proper CPUs. This has negative effects on both runtime predictability and schedulability analysis. Historically, the NVIDIA driver has configured GPUs to share interrupt identifiers. However, since late 2008, the GPL layer of the NVIDIA driver includes a compile-time option to enable modern interrupt handling mechanisms (NVIDIA, 2014a), specifically, Message Signaled Interrupts (MSI) (PCI-SIG, 2010).<sup>13</sup> Under MSI, interrupts are delivered “in-band” through the PCIe data pathways. Each GPU is assigned a unique interrupt identifier under MSI, allowing us to direct GPU interrupts to the appropriate CPUs. We note that MSI is enabled by default in the GPL layer of the NVIDIA driver, starting in late 2013 (NVIDIA, 2014a).

**GPU Bottom-Halves.** As we discuss in greater depth in Section 3.3, GPUSync is implemented within the Linux-based LITMUS<sup>RT</sup> kernel. GPUSync introduces a new class of LITMUS<sup>RT</sup>-aware daemons called `klmirqd`. This name is an abbreviation for “Litmus softirq daemon” and is prefixed with a “k” to indicate that the daemon executes in kernel space. `Klmirqd` daemons may function under any LITMUS<sup>RT</sup>-supported JLFP scheduling algorithm.

We associate one dedicated `klmirqd` daemon with each GPU. Each daemon processes the Linux tasklets (*i.e.*, bottom-halves) issued by GPU ISRs (*i.e.*, top-halves). These daemons may execute within the CPU cluster(s) of the tasks that use the associated GPU. Each inherits the maximum *effective* priority of any *suspended* job that is blocked waiting for a GPU operation of the associated GPU to complete. In effect, each tasklet is scheduled under this inherited priority. If there are no suspended jobs waiting for the daemon’s assigned GPU to complete an operation, then the daemon is scheduled with a base priority statically below that of any real-time task. This allows the daemon to take advantage of CPUs left idle.

In order to properly schedule bottom-halves, we must identify: (i) the GPU associated with each bottom half; (ii) the `klmirqd` daemon associated with each GPU; and (iii) the set of suspended jobs waiting for a

---

<sup>13</sup>MSI is enabled in the NVIDIA driver by asserting the `__NV_ENABLE_MSI` compile-time option.



The `klmirqd` daemon may inherit the suspended job's effective priority until the suspended task is ready to run.

### Step C: Intercept GPU Tasklet

The closed-source GPU driver must interface with the open-source Linux-based kernel. We exploit this fact to intercept tasklets dispatched by the driver. This is done by modifying the standard internal Linux `tasklet_schedule()` function. This function is used to issue a tasklet to the kernel for execution.

When `tasklet_schedule()` is called, the callback entry point of the deferred work is specified by a function pointer. We identify a tasklet as belonging to the closed-source GPU driver if this function pointer points to a memory region allocated to the driver. It is possible to make this determination, since the driver is loaded as a module (or kernel plugin). We inspect every callback function pointer of every dispatched tasklet, online, using Linux's module-related routines.<sup>14</sup> Thus, we alter `tasklet_schedule()` to intercept tasklets from the GPU driver and override their scheduling. It should be possible to use this technique to schedule tasklets of *any* driver in Linux that is loaded as a module, not just GPU drivers.

### Step D: Extract GPU Identifier

Merely intercepting GPU tasklets is not enough if a system has multiple GPUs; we must also identify which GPU raised the initial interrupt in order to determine which `klmirqd` daemon should handle the tasklet. It may be possible to perform this identification process at the lowest levels of interrupt handling (*i.e.*, when the OS looks up the appropriate ISR). However, this information must be passed into the deeper interrupt handling layers, potentially requiring invasive changes to the OS's internal APIs and the users of those APIs. Instead, we opt for a simpler solution closer to tasklet scheduling.

The GPU driver attaches a *memory address* to each tasklet, providing input parameters for the tasklet callback. This address points to a data block that contains a device identifier indicating which GPU raised the interrupt. However, locating this identifier within the data block is challenging since it is packaged in a driver-specific format.

Offline, we inspect the code of the driver's GPL layer to reverse engineer the memory address offset of the GPU identifier from the address that is attached to the tasklet. From code analysis, we find that

---

<sup>14</sup>This may sound like a costly operation, but it is actually quite a low-overhead process.

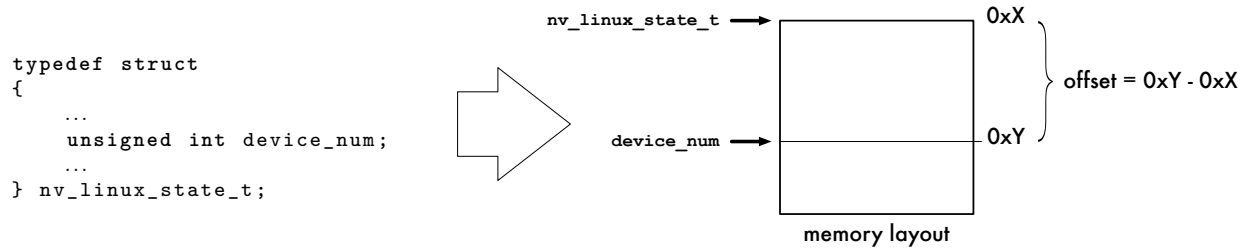


Figure 3.8: Memory layout of `nv_linux_state_t`.

the attached address is a pointer to a C-struct with the type “`nv_linux_state_t`.” Embedded within this struct is a 32-bit unsigned integer variable named “`device_num`.”<sup>15</sup> Through experimentation, we learn that this variable indicates the GPU that raised the interrupt. By modeling the memory layout of `nv_linux_state_t`, we can determine the offset needed to locate `device_num` within the tasklet’s data block. This is depicted in Figure 3.8.

From experience, we find that the memory layout of `nv_linux_state_t` may change with each new version of the NVIDIA driver. Moreover, several compile-time options of the GPL layer can change the layout of `nv_linux_state_t`, which may change the offset of `device_num`. Thus, a system designer must repeat this reverse engineering process with each new version of the NVIDIA driver to determine the proper offset. However, it may be feasible to automate this process by modifying the compilation scripts of the NVIDIA driver.

### Step E: Dispatch Tasklet

After completing Steps C and D, `tasklet_schedule()` passes intercepted GPU tasklets to the `klmirqd` daemon by calling `klmirqd_tasklet_schedule()`. This function takes the tasklet and `klmirqd` task identifier as arguments. The function inserts the tasklet into a queue of pending tasklets for the `klmirqd` daemon. The daemon is awoken if it is suspended waiting for work. Steps A and B ensure that the daemon is scheduled with the proper priority.

**GPU “Bottom-Bottom-Halves.”** In Section 2.2.3.1, we discussed how tasklets may never block on I/O or suspend from the CPU, since tasklets *may* be executed within interrupt context. In order to carry out such operations, a tasklet itself may defer additional “work items” to Linux `kworker` daemons. In a sense, these

<sup>15</sup>In recent versions of the GPL driver, this field has been renamed “`minor_num`.”

work items are “bottom-bottom-halves.” As with tasklets, we must also ensure that work items are properly scheduled.

We apply the same methodology we use to intercept GPU tasklets to intercept GPU work items. This is done by modifying the standard internal Linux `schedule_work()` function, which is normally used to pass work to the kworker daemons. Each work item contains callback and attached memory addresses, similarly to a tasklet—we identify GPU work items and associated GPU identifiers in the same way. Intercepted GPU work items may be passed to a *secondary* klmirqd daemon dedicated to work-item processing, or it may be passed to the klmirqd daemon that also processes tasklets. In the former case, we apply the same mechanisms we use in Steps A and B above to ensure the daemon is scheduled with the proper priority. In the latter case, the klmirqd daemon executes a single pending work item, if one exists, after executing each tasklet. This essentially fuses the tasklet and work item into a single unit of work. GPUSync supports both configurations. While the use of separate tasklet and work-time klmirqd daemons may improve performance through parallelism, it breaks the sporadic task model since the two daemons may execute concurrently under the same inherited priority.

### 3.2.5.2 CUDA Runtime Callback Threads

As we discussed in Sections 2.4.1 and 2.4.5, the CUDA runtime employs callback threads, one per GPU, to signal (wake) user threads that have suspended from execution while waiting for GPU operations to complete. These threads must be properly scheduled to avoid unbounded priority inversions. However, applying a real-time scheduling policy to these threads is challenging, since the threads are created and managed by closed-source software. Nonetheless, in our approach, we make no assumption of when the CUDA runtime creates callback threads. We also assume that we do not know which callback threads are associated with which GPUs. We now describe how GPUSync schedules the callback threads of the CUDA runtime.

A GPU-using task makes a system call, `set_helper_tasks()`, to GPUSync during the task’s initialization phase. The system call takes two flag parameters: `CURRENT` and `FUTURE`. If the flag `CURRENT` is set, then the OS examines each thread within the process of the calling task. Any encountered non-real-time thread is assumed to be a CUDA runtime callback thread. We apply a specialized LITMUS<sup>RT</sup> scheduling policy to each of these threads. If the flag `FUTURE` is set, then we automatically apply the same specialized scheduling policy to any threads created (forked) in the future.

The specialized scheduling policy we apply to callback threads is very similar to the one we use for `klmirqd` daemons. When job  $J_i$  suspends waiting for a GPU operation to complete, *all* callback threads of task  $T_i$  inherit the effective priority of  $J_i$ . This policy is enacted by the CPU scheduler when a task suspends while holding an engine lock—this is the same mechanism we used to enable priority inheritance for `klmirqd`. Normally, the simultaneous inheritance of  $J_i$ 's effective priority by multiple threads would break the sporadic task model, which assumes tasks are single threaded. However, it is safe in this particular instance because a job may only use one GPU at a time under GPUSync—only *one* callback thread will ever need to execute at a given moment. No two callback threads inheriting the same priority will ever execute simultaneously, so the sporadic task model is not violated.

### 3.3 Implementation

In this section, we discuss the implementation of several key components in GPUSync. We have already discussed several implementation-related issues that relate to the integration of GPUSync with LITMUS<sup>RT</sup> and the broader Linux kernel. We provide additional implementation details herein. We begin with general information on the implementation of GPUSync. We then discuss the challenges of implementing priority inheritance for the nested locking structure of GPUSync, while accommodating dynamic behaviors of GPUSync. This is followed by a discussion of the plugin infrastructure supported by the GPU Allocator to implement affinity-aware heuristics, such as the ones we discussed in Section 3.2.3.2. We conclude this section with a description of the GPUSync user interface used by real-time applications to communicate GPU resource requests to GPUSync.

#### 3.3.1 General Information

We implemented GPUSync as an extension to LITMUS<sup>RT</sup>, version 2014.1, which is based upon the 3.10.5 Linux kernel.<sup>16</sup> GPUSync adds approximately 20,000 lines of code to LITMUS<sup>RT</sup>. Contributions to this total by category are approximately: GPU Allocator and locking protocols, 35%; scheduler enhancements, budgeting, and nested inheritance, 35%; GPU interrupt and callback thread management, 20%; miscellaneous infrastructural changes, 10%. For comparison, the LITMUS<sup>RT</sup> patch to the Linux 3.10.5 kernel is roughly 15,000 lines of code. The source code for GPUSync is available at

---

<sup>16</sup>We distribute GPUSync as open source under the GNU General Public License, version 2. The code is currently available at [www.github.com/GElliott](http://www.github.com/GElliott).

LITMUS<sup>RT</sup> provides a plugin-based real-time scheduling framework within the Linux kernel, where particular real-time scheduling algorithms are implemented as plugins. With the exception of modifications to the tasklet and work-item processing of Linux (see Section 3.2.5), GPUSync is implemented almost entirely within LITMUS<sup>RT</sup>—GPUSync rarely interfaces directly with Linux kernel components.

We limit the integration of GPUSync with LITMUS<sup>RT</sup> to LITMUS<sup>RT</sup>'s C-EDF plugin. This plugin can be configured to cluster CPUs along different boundaries of the hardware memory hierarchy. For example, if we cluster around CPU private caches (*e.g.*, the L1), then our C-EDF scheduler is equivalent to P-EDF. Likewise, if we cluster around main memory, our C-EDF scheduler is equivalent to G-EDF. The LITMUS<sup>RT</sup> C-EDF plugin provides us with the necessary flexibility to test a variety of CPU cluster configurations with a single code base. We recognize that this flexibility may result in slightly greater system overheads. For example, the LITMUS<sup>RT</sup> P-EDF plugin is more streamlined than its C-EDF counterpart (even when similarly configured), since the uniprocessor nature of P-EDF admits assumptions that reduce code complexity. Finally, we note that the C-EDF plugin requires only minor modifications to support fixed-priority scheduling, as the programmer need only provide a new prioritization function for comparing the priority of two tasks. We extended the LITMUS<sup>RT</sup> scheduler plugin API to expose the prioritization function used by a scheduler to the GPU Allocator and engine locks, so changes in the prioritization function are transparent to GPUSync components.

### 3.3.2 Scheduling Policies

Threads in Linux are generally scheduled under one of two primary policies: `SCHED_OTHER` or `SCHED_FIFO`.<sup>17</sup> The `SCHED_OTHER` policy is used to schedule general purpose, non-real-time, applications. `SCHED_FIFO` is used to schedule fixed-priority real-time tasks in accordance to the POSIX standard. Linux prioritizes `SCHED_FIFO` threads above `SCHED_OTHER` threads, as illustrated in Figure 3.9(a). LITMUS<sup>RT</sup> introduces a third scheduling policy: `SCHED_LITMUS`. All threads scheduled by the `SCHED_LITMUS` policy have a greater priority than `SCHED_FIFO` threads. This is depicted in Figure 3.9(b).

The GPUSync implementation splits the `SCHED_LITMUS` scheduling policy into three sub-policies. We use sub-policies within LITMUS<sup>RT</sup> as it allows us to quickly transition threads among them. In decreasing static priority, these sub-policies are: `normal`, `callback`, and `daemon`. Conventional real-time tasks are

---

<sup>17</sup>Linux provides additional scheduling policies, such as `SCHED_BATCH` and `SCHED_RR`. These are similar to `SCHED_OTHER` and `SCHED_FIFO`, respectively. We do not discuss them in order to simplify the presentation.

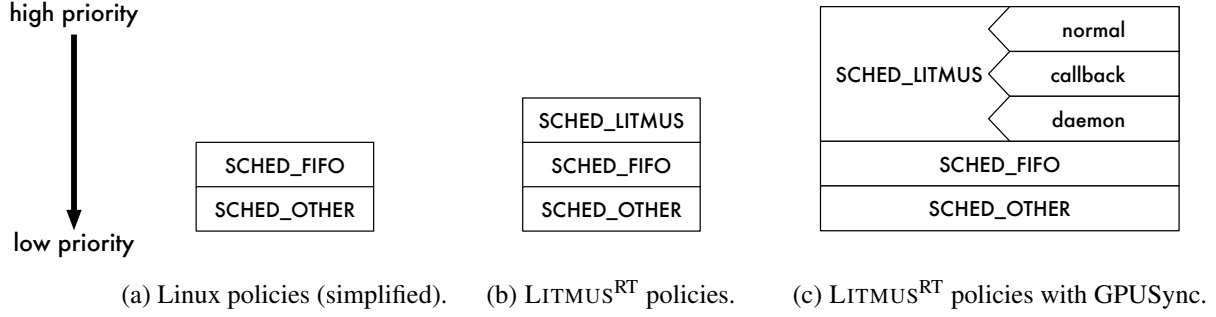


Figure 3.9: Relative static priorities among scheduling policies.

scheduled under the normal sub-policy. CUDA runtime callback threads are scheduled under the callback sub-policy. Klmirqd threads are scheduled under the daemon sub-policy. We prioritize the callback sub-policy over the daemon sub-policy in an effort to expedite completion of each GPU operation. CUDA callback and klmirqd threads transition to the normal sub-policy when they inherit a priority from a normal real-time task. Figure 3.9 depicts the relative static priorities of each scheduling policy.

Threads that share the same policy compete for CPU time in accordance to the policy’s CPU scheduler. For example, we may configure LITMUS<sup>RT</sup> to schedule SCHED\_LITMUS-policy tasks with C-EDF. Jobs in LITMUS<sup>RT</sup> may have equal priorities. For instance, the deadlines of two jobs may coincide under EDF scheduling. LITMUS<sup>RT</sup> implements several configurable mechanisms to break such ties in priority. For GPUSync, we configure LITMUS<sup>RT</sup> to use tie-break heuristics that fairly distribute deadline lateness, proportional to each task’s relative deadline, among tasks. This heuristic can have a significant effect on the response time of jobs on a temporarily overloaded system. For callback and klmirqd threads that do not actively inherit a priority from a normal LITMUS<sup>RT</sup> task, we break ties in a manner that fairly distributes CPU time among them.

### 3.3.3 Priority Propagation

The two-level nested locking structure of GPUSync requires the propagation of inherited or donated priorities along complex chains of dependent tasks.

Figure 3.10 illustrates a possible chain of task dependencies in GPUSync. In this example, suppose all tasks are scheduled within the same CPU and GPU clusters. Here, task  $T_1$  is in the DQ of the GPU Allocator and donates its priority to task  $T_2$  with a request enqueued in  $FQ_0$  of the GPU Allocator. Task  $T_3$  holds the token of  $FQ_0$ , so it may inherit the effective priority of  $T_2$ .  $T_3$  has an unsatisfied request for the first CE of



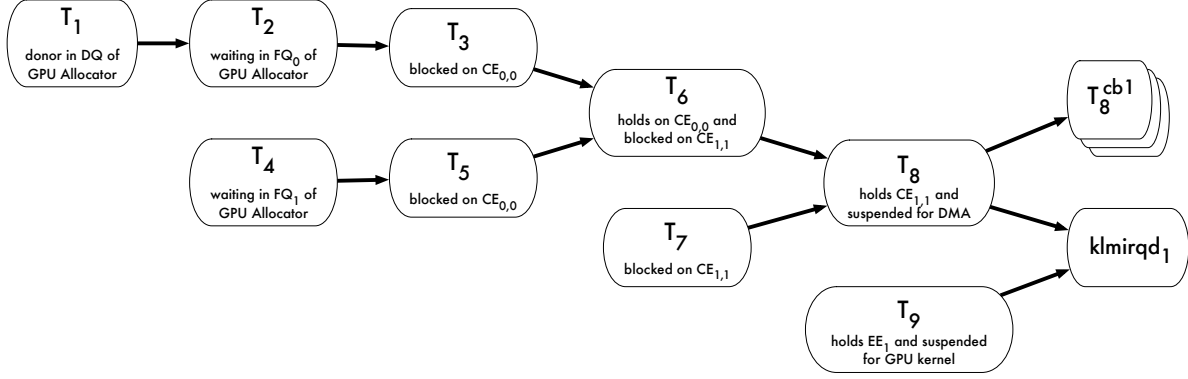


Figure 3.10: Example of a complex chain of execution dependencies for tasks using GPUSync.

GPU<sub>0</sub>, or CE<sub>0,0</sub>. Another task  $T_4$  is enqueued on FQ<sub>1</sub> in the GPU Allocator. Let us assume that the token of FQ<sub>1</sub> is associated with GPU<sub>0</sub>. Task  $T_5$  holds the token of FQ<sub>1</sub>, so it may inherit the effective priority of  $T_4$ .  $T_5$  also has an unsatisfied request for CE<sub>0,0</sub>. Task  $T_6$  issued a DGL request for both CE<sub>0,0</sub> and CE<sub>1,1</sub>, prior to the CE requests of  $T_3$  and  $T_5$ , in order to perform a peer-to-peer migration between GPU<sub>0</sub> and GPU<sub>1</sub>.  $T_6$ 's request for CE<sub>0,0</sub> has been satisfied, but the task remains blocked waiting for CE<sub>1,1</sub>. Because  $T_6$  holds the engine lock of CE<sub>0,0</sub>, it may inherit either the effective priority of  $T_3$  or  $T_5$ . Task  $T_7$  has an unsatisfied request for CE<sub>1,1</sub>. Task  $T_8$  holds the engine lock for CE<sub>1,1</sub>, so it may inherit either the effective priority of  $T_6$  or  $T_7$ .  $T_8$  has issued its DMA operation to CE<sub>1,1</sub>. The DMA is incomplete, so  $T_8$  is suspended. Thus, all of the callback threads of  $T_8$  may inherit the effective priority of  $T_8$ , including the callback thread for GPU<sub>1</sub>,  $T_8^{cb1}$ . Task  $T_9$  holds the engine lock for EE<sub>1</sub> and is waiting for a GPU kernel to complete, so  $T_9$  has also suspended. Finally, the klmirqd daemon for GPU<sub>1</sub>, klmirqd<sub>1</sub>, may inherit the effective priority of either  $T_8$  or  $T_9$ , since both tasks have incomplete operations on GPU<sub>1</sub> and both tasks are suspended. What is the effective priority of  $T_8^{cb1}$ ? It is the maximum priority among tasks  $T_1$  through  $T_8$ . What is the effective priority of klmirqd<sub>1</sub>? It is the maximum priority among tasks  $T_1$  through  $T_9$ .

Implementing the necessary mechanism to propagate effective priorities through dependency chains such as the one in Figure 3.10 is challenging since information about each task is distributed among several data structures (*e.g.*, the request queues of the various locks). We may imagine a recursive algorithm that propagates the donated priority of  $T_1$  down the dependency chain of Figure 3.10 when  $T_1$  issues its token—indeed, such an algorithm is obviously necessary. Under conventional JLFP locking protocols, the implementation of such an algorithm is straight forward due to the following two invariants:

- I1** The effective priority of a job may only increase monotonically, unless such a job releases a held lock.

**I2** The base priorities of jobs are fixed.

However, neither of these hold under certain configurations of GPUSync, so implementation is more complex. Let us consider two cases where GPUSync breaks conventional Invariants I1 and I2.

In the first case, the request stealing rule (Rule R5 of the R<sup>2</sup>DGLP) used by the GPU Allocator may break Invariant I1, since the GPU Allocator may move a pending request from one FQ to another. Suppose task  $T_4$  in Figure 3.10 has the highest base priority among all tasks, so  $T_8^{cb_1}$  and  $\text{klmirqd}_1$  inherit the effective priority of  $T_4$ . If  $T_4$  is moved to another FQ, say,  $\text{FQ}_2$ , then we must determine a new effective priorities for  $T_8^{cb_1}$  and  $\text{klmirqd}_1$ .  $T_8^{cb_1}$  may draw an effective priority from the tasks  $T_1$  through  $T_8$  (except  $T_4$ ).  $\text{klmirqd}_1$  may draw an effective priority from the same tasks, in addition to task  $T_9$ . Whatever these new effective priorities may be, they are less than the base priority of  $T_4$  (which remains unchanged). Thus, the effective priorities of  $T_8^{cb_1}$  and  $\text{klmirqd}_1$  decrease, breaking Invariant I1.

In the second case, GPUSync's budget enforcement policies can cause the base priority of a job to change under deadline-based schedulers, violating Invariant I2. This is clear under the early-releasing policy. However, let us discuss a more complicated scenario under the BWI policy. With out loss of generality, let us assume that the tasks in Figure 3.10 are synchronous (*i.e.*, each task releases a job at the same time instant), periodic, and have implicit deadlines. Thus, the  $j^{\text{th}}$  jobs of all tasks share the same absolute deadline, since they share a common release time and relative deadline. Jobs are prioritized by EDF and ties are broken by task index such that  $T_i$  has greater priority than  $T_{i+1}$ . Suppose job  $J_{8,j}$  of task  $T_8$  overruns its provisioned budget.  $J_{8,j}$  drains the budget of job  $J_{1,j}$  first since  $J_{1,j}$  has the highest priority. (Job  $J_{i,j}$  has greater priority than job  $J_{i+1,j}$  due to tie-breaking.) Once exhausted, the priority of  $J_{1,j}$  is decreased through early-releasing and now has the lowest priority among all jobs, since  $D_{1,j+1} > D_{i,j}$  for  $i \in \{2, 3, \dots, 9\}$ . As a result, we must determine a new effective priority for all other jobs that inherited the priority of  $J_{1,j}$ . The job  $J_{2,j}$  now has the highest priority among all jobs, so  $J_{8,j}$  begins consuming the budget of  $J_{2,j}$ .  $J_{8,j}$  continues to exhaust the budgets of jobs  $J_{2,j}$  through  $J_{7,j}$ , until  $J_{8,j}$  begins to consume its own budget. The base priority of each job decreases as its budget is exhausted, breaking Invariant I2.

We describe the situations that break Invariants I1 and I2 not because they imply a fundamental departure from a recursive algorithm to propagate priorities through chains of dependent tasks, but rather to illustrate the fact that effective priorities of tasks under GPUSync may be in flux. The core algorithm to propagate effective priorities among tasks is embodied by a three-step recursive process where we *detect* a potential

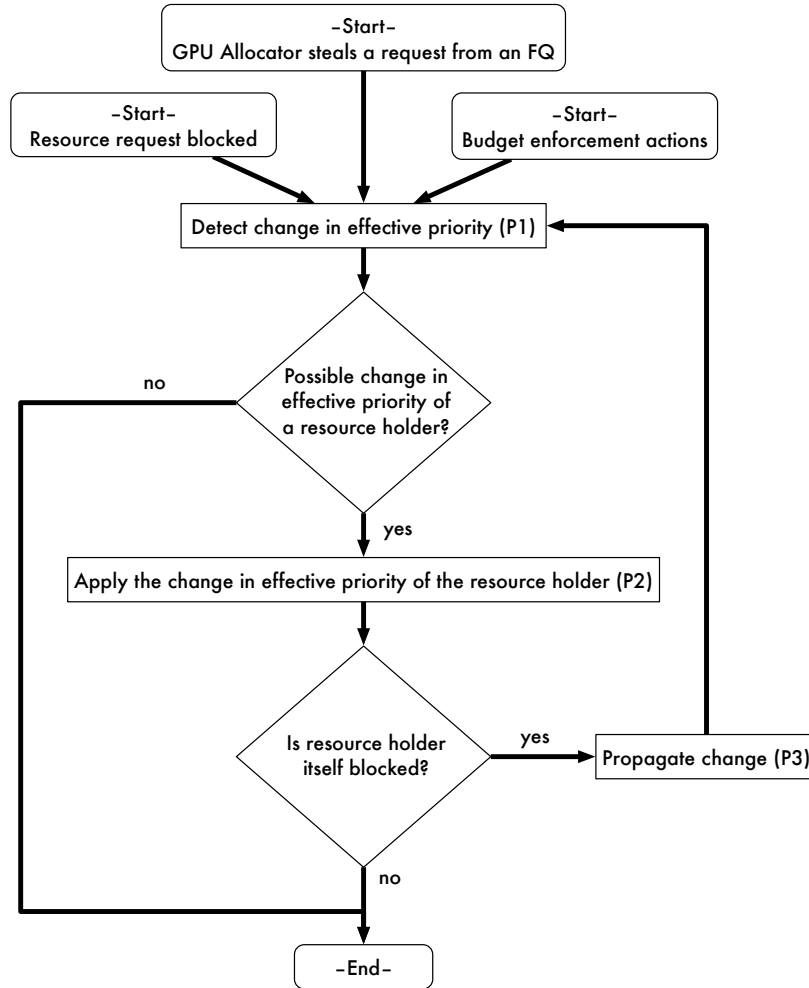


Figure 3.11: Recursive algorithm to propagate changes in effective priority.

change in the effective priority of a task, *apply* the change if necessary, and *propagate* that change if the task is itself blocked for a resource. We summarize these steps with the flowchart depicted in Figure 3.11. We now describe the algorithm we use to propagate effective priorities in more detail. We label each set of operations that make up each step in our propagation algorithm.

### P1: Detect

For every protected resource (token or engine lock), the implementation of the locking protocol tracks the effective priorities of all pending requests from which a resource holder may inherit. This is done by using a max-heap, ordered by effective priority. Consider the implementation of a FIFO-ordered engine lock. In addition to a FIFO queue to order requests, a heap is populated with the unsatisfied

requests. We may efficiently determine the greatest effective priority among the unsatisfied requests by examining the root node of the heap. Updates to the heap are logarithmic in runtime complexity.

## **P2: Apply**

Each task may inherit an effective priority through any one of several locks it may hold concurrently. Within the Linux kernel `task_struct` of each task, we maintain a max-heap of the effective priorities that the task *might* inherit. The nodes in this heap reference the root nodes of the associated max-heaps maintained by P1. A task inherits the effective priority of the root node of the `task_struct`-heap if the task has a lower base priority. We update `task_struct`-heap whenever the root node of an associated heap maintained by P1 changes. Concurrent updates to the `task_struct`-heap are serialized by a per-task spinlock.

Additional measures are taken to propagate priorities to `klmirqd` daemons and callback threads. The `klmirqd` daemon uses the `task_struct`-heap to determine its effective priority. However, instead of referencing root nodes of the heaps maintained by P1, the nodes reference tasks suspended waiting for GPU operations to complete. If the effective priority of a task changes while it is suspended holding an engine lock, then we propagate the change to the `klmirqd` daemon assigned to the GPU for the token held by the task and reevaluate the daemon's effective priority. For callback threads, if the effective priority of a task changes while it is suspended holding an engine lock, then the callback threads of the task inherit the new effective priority.

## **P3: Propagate**

Within each `task_struct`, we store a pointer to a lock for which a task is blocked.<sup>18</sup> The value of this pointer is null if the task is not blocked. If the effective priority of a task changes while it is blocked, we update the max-heap maintained by P1. Changes are recursively propagated.

We take the following actions to support the request stealing rule used by the GPU Allocator. First, we remove the request to steal from  $FQ_x$ . We then update the max-heap used by the GPU Allocator to track the effective priorities of requests in  $FQ_x$ . We apply the potential change in the effective priority of the token holder of  $FQ_x$  by following P1 and any subsequent steps, as necessary.

---

<sup>18</sup>When DGLs are used, this pointer points to any single requested lock not held by the task.

We follow a multi-step process if the budget of a job is exhausted when GPUSync is configured to use BWI and dynamic-priority JLFP scheduling (*e.g.*, EDF):

1. If the job is blocked for a token, we coordinate with the GPU Allocator to abort the token request. This may result in changes to a max-heap maintained by the GPU Allocator. We apply any potential change in the effective priority of the token holder by following P1 and any subsequent steps, as necessary.
2. We decrease the base priority of the job through early-releasing.
3. If the job holds a resource lock, then we reevaluate the effective priority of the job by following P1 and any subsequent steps, as necessary. This step is required since the job may need to inherit a priority due to its decreased base priority.
4. If we aborted a token request above, then we re-issue the request. This may result in changes to a max-heap maintained by the GPU Allocator—changes in effective priorities are propagated.

We only perform steps 2 and 3 under the early-releasing policy under dynamic-priority JLFP scheduling.

We conclude this section with a remark on the challenges of implementing the above algorithms. These algorithms require coordination among software components that implement locking protocols, schedulers, and budget enforcement policies. Each component may use a variety of spinlocks to protect kernel data structures. The implementation must be extremely careful with respect to when, and in what order, these locks are acquired so that deadlock is avoided.

### 3.3.4 Heuristic Plugins for the GPU Allocator

We use a plugin-based software architecture for the GPU Allocator to implement the heuristics described in Section 3.2.3.2. This provides a clean separation between the core GPU Allocator locking protocol algorithm ( $R^2$ DGLP or CK-OMLP), the heuristics, and other software components of GPUSync. The design also facilitates experimentation with new heuristics. A *heuristic plugin* defines a collection of heuristics used to guide the actions of the GPU Allocator. We may use different plugins to implement a variety of strategies such as “optimize the average case” or “aggressive migration.” We implement the heuristics discussed in Section 3.2.3.2 as a single plugin.

Table 3.1 lists the heuristic plugin API. The advisory functions are invoked by the GPU Allocator to request guidance from the plugin at key algorithmic decision points. For example, the GPU Allocator may

Advisory API	
<code>advise_enqueue()</code>	Recommend the FQ on which to enqueue a given request.
<code>advise_steal()</code>	Recommend a request to steal from an FQ.
<code>advise_donee_selection()</code>	Recommend a donee for a priority donor.
Notification API	
<code>notify_enqueue()</code>	A request has been enqueued on a specified FQ.
<code>notify_dequeue()</code>	A request has been removed from a specified FQ.
<code>notify_acquired()</code>	A specified token has been acquired by a specified job.
<code>notify_freed()</code>	A specified token has been freed by a specified job.
<code>notify_exit()</code>	A given task has terminated.

Table 3.1: Heuristic plugin interface for the GPU Allocator.

call `advise_enqueue()` to solicit advice from the plugin in deciding which FQ a given request should be enqueued—we use this interface to implement Heuristics H1 and H2. The GPU Allocator is free to *reject* a recommendation made by any advisory function. We do so in order to protect the real-time correctness of the underlying GPU Allocator locking protocol algorithm from poor advice provided by the plugin.

The notification functions are invoked by the GPU Allocator to inform the plugin of the Allocator’s actions. Heuristic plugins use these functions to maintain their own internal state. For instance, we use `notify_enqueue()` and `notify_dequeue()` to maintain a cached estimate of the lengths of each FQ (*i.e.*, Equation (3.2) of Section 3.2.3.2, which is necessary to efficiently implement Heuristic H2). Also, we use the function `notify_exit()` to inform a plugin that a real-time task has terminated. This information is needed to properly implement Heuristic H1, which attempts to distribute sporadic tasks among GPUs.

We also take advantage of the notification functions to interface with other software components. For instance, we use `notify_acquired()` and `notify_freed()` to bridge the GPU Allocator and the GPU Registry that is used for scheduling `klmirqd` daemons (Section 3.2.5.1). We also use these functions to integrate the GPU Allocator and the Cost Predictor. When the GPU Allocator calls `notify_acquired()`, the plugin begins tracking the combined CPU and GPU execution time of a job within the token critical section, as described in Section 3.2.3.3. Similarly, the plugin provides the Cost Predictor with a new observation when the GPU Allocator calls `notify_freed()`.

### 3.3.5 User Interface

The developers of `LITMUSRT` provide a companion user-space library, called *liblitmus*, to ease the development of real-time `LITMUSRT` tasks. The *liblitmus* API provides functions for creating and initializing real-time tasks, assigning tasks to clusters, and job management.

We extend liblitmus to provide the user with an interface to GPUSync.<sup>19</sup> This extended library does *not* serve as a GPGPU interception library (Section 3.1.2). Rather, such an interception library would *use* the services exposed by our GPUSync-modified liblitmus to enact scheduling policies. The extended library includes functions for creating and configuring the GPU Allocator and engine locks. The API to create an instance of a GPU Allocator requires the callee to specify which GPUs are to be managed by the allocator and values for  $p$  (the maximum number of per-GPU concurrent users) and  $f$  (the maximum length of each FQ). The callee may also specify which, if any, heuristic plugin should be employed by the GPU Allocator instance. APIs for creating and configuring engine locks are also provided. Each GPU Allocator and engine lock instance is given a unique name by the creating process. Other real-time tasks use these names to obtain the necessary references to the created objects.

Additional extensions to the liblitmus API include functions for obtaining GPU tokens, locking and unlocking engine locks (with or without the use of DGLs), and performing chunked memory copies. There are also routines to facilitate exception-based handling of budget exhaustion signals.

### 3.4 Conclusion

This concludes our description of the design and implementation of GPUSync. Through the careful consideration of tradeoffs among a variety of real-time GPU scheduler designs, we have designed GPUSync to be an API-driven scheduler implemented within the operating system. We sacrifice the ability to strictly enforce all GPU scheduling decisions in order to support closed-source software. GPUSync is designed around a synchronization-based philosophy to real-time GPU scheduling. This provides us with a variety of tools and methodologies we need to ensure real-time predicability.

---

<sup>19</sup>This extended version of liblitmus is currently available at [www.github.com/GElliott](http://www.github.com/GElliott) under the GNU General Public License, version 2.

## CHAPTER 4: EVALUATION<sup>1</sup>

In this chapter, we evaluate real-time properties of GPUSync. Our evaluation is in two parts. In the first part, we develop a theoretical model of GPUSync for our evaluation platform, a twelve-core, eight GPU, system. This model incorporates carefully measured system overheads and overhead-aware schedulability analysis. Upon this model, we perform a large-scale set of experiments where we evaluate the schedulability of randomly generated task sets under a variety of GPUSync configurations. Although not exhaustive, this evaluation is broad. It required over 85,000 CPU hours to complete, testing over 2.8 billion task sets for schedulability. We show that real-time guarantees differ greatly among GPUSync configurations. In the second part of our evaluation, we investigate the runtime performance of GPUSync. We first examine the efficacy of GPUSync’s budget enforcement mechanisms, the accuracy of GPUSync’s Cost Predictor, and the ability of GPUSync’s affinity-aware GPU token allocation heuristics to reduce costly GPU migrations. We then evaluate a variety of GPUSync configurations through additional runtime experiments. Here, we execute task sets made up of tasks that execute GPU-based computer vision algorithms. We report upon the differences in observed real-time performance among the GPUSync configurations.

The remainder of this chapter is organized as follows. In Section 4.1, we describe the evaluation platform upon which our schedulability and runtime evaluations are based. We then describe the platform configurations (*e.g.*, CPU and GPU cluster configurations) we consider in our evaluations in Section 4.2. In Section 4.3, we develop our real-time model for evaluating task set schedulability under GPUSync. This section includes a description of the methods used to gather empirical data of GPU-related system overheads—we present this data as well. Section 4.3 also includes detailed blocking analysis of GPUSync’s engine and token locks, and the methodology used to integrate GPU-related overheads into schedulability analysis. In Section 4.4, we describe our runtime evaluation of GPUSync, and we present our results. We conclude in

---

<sup>1</sup> Portions of this chapter previously appeared in conference proceedings. The original citations are as follows:

Elliott, G., Ward, B., and Anderson, J. (2013). GPUSync: A framework for real-time GPU management. In *Proceedings of the 34th IEEE International Real-Time Systems Symposium*, pages 33–44;

Elliott, G. and Anderson, J. (2014). Exploring the multitude of real-time multi-GPU configurations. In *Proceedings of the 35th IEEE International Real-Time Systems Symposium*, pages 260–271.



Section 4.5, where we compare and contrast the results of our schedulability experiments with the empirical results of our runtime evaluation.

## 4.1 Evaluation Platform

We evaluate our implementation of GPUSync on a high-end multicore, multi-GPU, platform. This platform has two NUMA nodes, each like the system depicted in Figure 2.20. Each NUMA node is equipped with one Xeon X5060 processor with six 2.67GHz cores, and four NVIDIA K5000 Quadro GPUs. In total, our evaluation platform is equipped with twelve CPUs and eight GPUs. We first provide additional details about our CPUs before describing our GPUs. Each CPU core has a private 32KB L1 cache for instructions, and another of the same size for data. Each CPU core also has a private 256KB L2 cache. The six cores on each X5060 processor share a single 12MB L3 cache. On our platform, the L3 is an *inclusive* cache, meaning that it contains copies of the data stored in caches above it. Each Quadro K5000 GPU connects to the platform through PCIe 2.0, using 16 PCIe lanes. The K5000 has two CEs, so each GPU is capable of simultaneous bi-directional DMA operations. This GPU also supports peer-to-peer DMA. We call peer-to-peer DMA operations between two GPUs that share the same PCIe switch “near” peer-to-peer DMA operations. Similarly, peer-to-peer DMA operations between two GPUs that share the same I/O hub, but not a PCIe switch, “far” peer-to-peer DMA operations. The hardware scheduler issues we discussed in Section 2.4.4 limit the K5000. For all of the evaluations performed herein, we took the steps necessary to ensure that GPU engines operated independently, maximizing the parallelism of our platform.

We justify our use of a high-end platform *in lieu* of a smaller embedded platform, such as those we discussed in Section 2.3, in two ways. First, the complexity of the high-end platform enables research into more complex scheduling problems (*e.g.*, clustered GPU scheduling). Second, as technology advances, embedded platforms often resemble earlier higher-end platforms. For example, NVIDIA recently announced plans to develop a platform targeted to computer vision processing in automotive applications (Ho and Smith, 2015). This platform uses two Tegra chips, each containing four CPU cores and an integrated GPU. Although details of this new product are currently unavailable, the high-level architecture of this platform appears to be not unlike that of our multi-GPU NUMA platform.

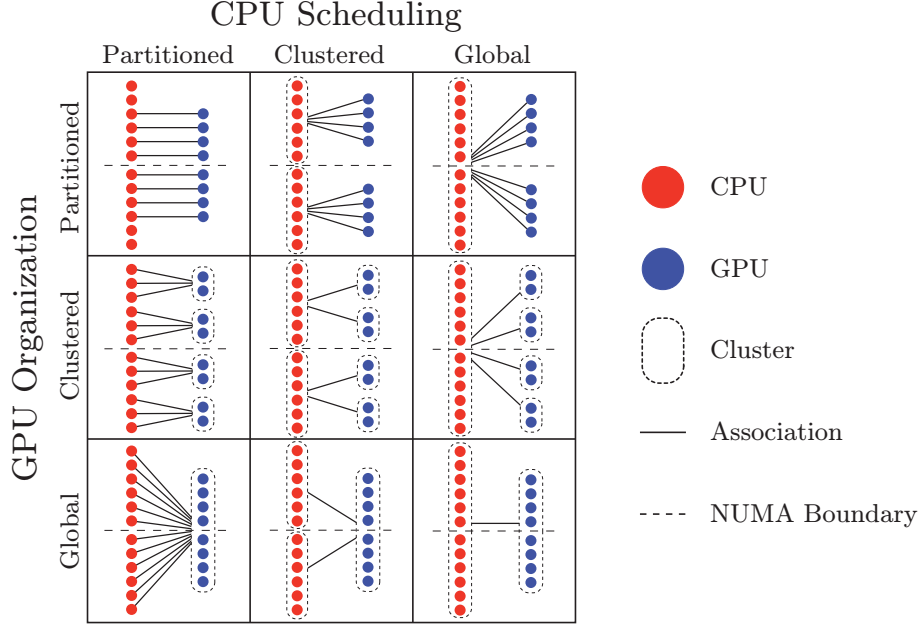


Figure 4.1: Concrete platform configurations.

## 4.2 Platform Configurations

As we described in Chapter 1 (see Figure 1.4), a multicore multi-GPU platform may be organized in a number of ways. We define a notational system to help us describe specific organizational configurations. We use a matrix of several configurations, depicted in Figure 4.1, for a twelve-CPU, eight-GPU platform to illustrate several examples. We refer to each cell in Figure 4.1 using a column-major tuple, with the indices  $P$ ,  $C$ , and  $G$  denoting partition, clustered, and global choices, respectively. The tuple  $(P, P)$  refers to the top-left corner—a configuration with partitioned CPUs and GPUs. Likewise,  $(G, C)$  indicates the right-most middle cell—globally scheduled CPUs with clustered GPUs. We use the wildcard  $*$  to refer to an entire row or column: *e.g.*,  $(P, *)$  refers to the left-most column—all configurations with partitioned CPUs. Within each cell, individual CPUs and GPUs are shown on the left and right, respectively. Dashed boxes delineate CPU and GPU clusters (no boxes are used in partitioned cases). The solid lines depict the association between CPUs and GPUs. For example, the solid lines in  $(C, C)$  indicate that two GPU clusters are wholly assigned to each CPU cluster. Finally, the horizontal dashed line across each cell denotes the NUMA boundary of the system.

When necessary, we extend our notation to denote the number of CPUs or GPUs within a cluster by using a subscript. For example,  $C_2$  may denote a cluster of two GPUs. We must also disambiguate between

GPU clusters where migration is carried out by peer-to-peer DMA and those where migration is carried out by DMA operations to and from system memory. To do so, we denote peer-to-peer configurations with the superscript “P2P.” For instance,  $C_4^{P2P}$  describes a cluster of four GPUs with peer-to-peer migration.

### 4.3 Schedulability Analysis

In this section, we develop a model of our evaluation platform for testing real-time schedulability. We begin by examining and measuring GPU-related overheads. We show that the use of GPUs can lead to significant system overheads in Section 4.3.1. Due to the large configuration space represented by the combination of GPUSync parameters, platform organizational choices, and variety of JLFP schedulers support by GPUSync, we scope our study to the subset of configurations we expect will show the most promise, while still broadly covering a range of possible configurations. We describe our scope and rationale in Section 4.3.2. In Section 4.3.3, we then discuss the task model we use to characterize sporadic task sets with tasks that use GPUs. We then develop detailed pi-blocking analysis for token and engine lock requests, which we must use in schedulability analysis in Section 4.3.4. In Section 4.3.5, we extend the overhead-aware preemption-centric schedulability analysis we discussed in Section 2.1.8 to incorporate our measured GPU-related overheads. Finally, we perform a broad set of schedulability experiments in Section 4.3.6. These experiments are meant to determine the most theoretically promising GPUSync configurations, and serve to show to what degree GPUs can increase computational capacity, despite heavy system overheads, under our model of GPUSync.

#### 4.3.1 Overhead Measurement

We now investigate and quantify overheads due to GPU processing. In general, we classify overheads as being algorithmic or memory-related. Algorithmic overheads are those due to code execution and event signaling. These include overheads due to thread context switching, scheduling, job release queuing, inter-processor interrupt latency, CPU clock tick processing, and interrupt processing. Memory overheads are those that increase execution time due to shared use of memory and data busses. These include overheads due to cache preemption/migration delays and memory bus contention. We already discussed non-GPU-related overheads from both categories in Sections 2.1.8 and 2.1.8.2 (and summarized in Tables 2.3 and 2.4). Overheads due to GPU processing also fall within algorithmic and memory-related categories. Namely, GPU

interrupt handling (algorithmic) and DMA operations (memory-related). The process for quantifying GPU interrupt handling overheads is relatively straightforward—we directly measure the execution time of GPU interrupt-handling routines within the OS. However, the quantification of DMA overheads requires a more nuanced approach, since the heavy load that DMA operations place on the system memory bus also affects code executing on the CPUs. We now present the methodology we used for measuring and quantifying both algorithmic and memory-related overheads.

#### 4.3.1.1 Algorithmic Overheads

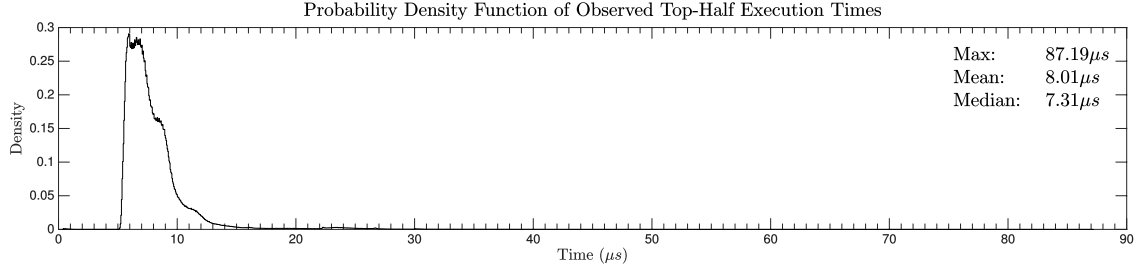
We measured algorithmic overheads using the lightweight tracing facilities of LITMUS<sup>RT</sup> while executing workloads that stress the various hardware components managed by GPUSync. Measurements were taken under different CPU and GPU cluster configurations, as well as with task sets of varying sizes (in order to capture overhead trends dependent upon the number of tasks). Over 11GB of trace data was recorded (a single trace event is only 16 bytes in size). We distilled this data into average and worst-case overheads for each of the algorithmic overheads in Tables 2.3 and 2.4. The properties of non-GPU-related overheads needed for preemption-centric accounting have been thoroughly studied by Brandenburg (2011b), so we will not replicate his work here. However, we do discuss algorithmic overheads related to GPUs.

GPUs interact with the host platform through I/O interrupts. As we discussed in Section 2.2.3, interrupt processing is split into “top” and “bottom” halves. Overhead-aware schedulability analysis requires that we quantify the execution cost of top and bottom halves of GPU interrupt processing. In order to do so, we stressed our evaluation platform with 30 GPU-using tasks that performed computer vision calculations on pre-recorded video streams.<sup>2</sup> Tasks were assigned periods between 33ms and 100ms. GPUSync allocated GPUs to jobs and arbitrated access to GPU engines. We instrumented the code paths of GPU top-half and bottom-half routines to measure their execution times. We took execution time measurements over a duration of 20 minutes.

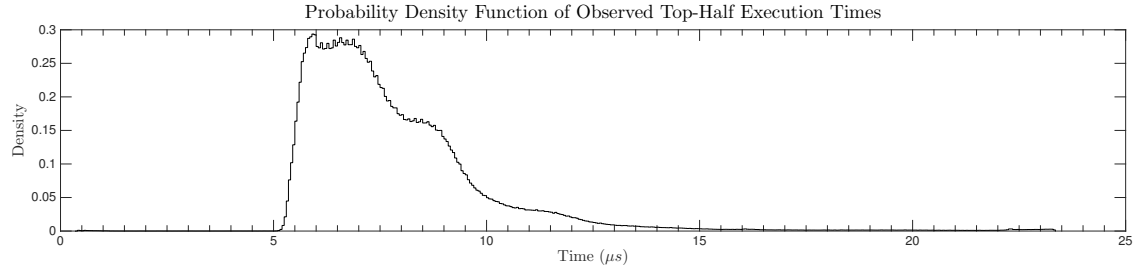
Figure 4.2(a) gives the probability density function (PDF) derived from over 4,500,000 GPU top-half execution time observations. In the PDF, to determine the probability that a given top-half execution time measurement falls within the domain  $[a, b]$ , we sum the area under the curve between  $x = a$  and  $x = b$ ; the total area under each curve is 1.0. The most striking aspects of this data are the outliers: the maximum value

---

<sup>2</sup>We describe this code in more detail later in Section 4.4.2.



(a) All measurements.



(b) Measurements below the 99<sup>th</sup> percentile.

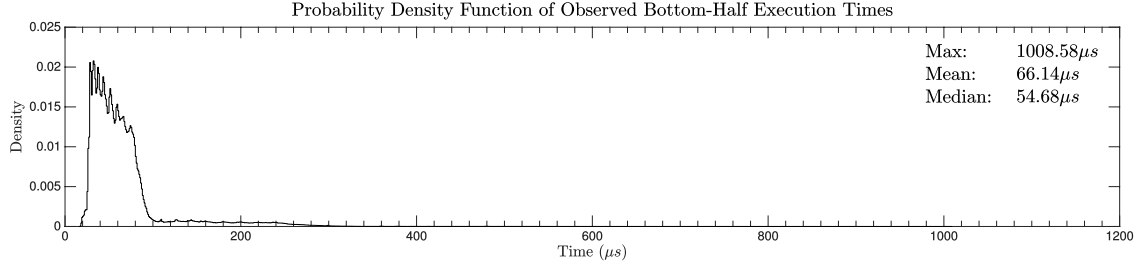
Figure 4.2: PDF of GPU top-half execution time.

is 87.19 $\mu$ s, yet the mean and median are only 8.01 $\mu$ s and 7.31 $\mu$ s, respectively. In order to better observe the shape of the PDF, we plot the same data in Figure 4.2(b), but we clip the domain to include only measurements below the 99<sup>th</sup> percentile. The four humps in the PDF (centered near  $x = 5.5\mu$ s,  $x = 6.5\mu$ s,  $x = 8.5\mu$ s, and  $x = 11\mu$ s, respectively) suggest that there may be at least four code paths commonly taken by the ISR.

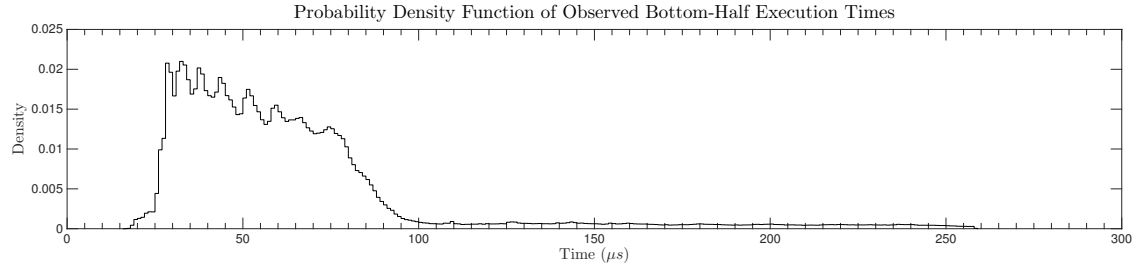
Figure 4.3(a) gives the PDF derived from about 4,200,000 GPU bottom-half execution time observations.<sup>3</sup> This PDF shares a similar characteristic with the PDF for top-halves in Figure 4.2(a): extreme outliers. Here, we see that the maximum value is 1008.58 $\mu$ s, yet the mean and median are only 66.14 $\mu$ s and 54.68 $\mu$ s, respectively. Moreover, the maximum-to-median ratio for bottom-half execution time is approximately 18.4 (1008.58/54.68  $\approx$  18.4). In contrast, this ratio is roughly 11.9 (87.19/7.31  $\approx$  11.9) for top-half execution time. In other words, the severity of outlier behavior in bottom-half execution time is worse. Figure 4.3(b) plots the same data, but we clip the domain to stop at the 99<sup>th</sup> percentile. In this figure, we observe at eight distinct humps in the PDF that span the approximate domain of [30 $\mu$ s, 80 $\mu$ s]; even if we ignore outlier behavior, there is still a great deal of variance in bottom-half execution time.

We now examine the outlier behavior of top-half and bottom-half execution time in more detail. Figure 4.4 depicts the *complement* cumulative distribution function (CCDF) of top-half and bottom-half observations, in

<sup>3</sup>The number of bottom-half observations is less than top-half observations. This implies that not every top-half spawns a corresponding bottom-half.



(a) All measurements.



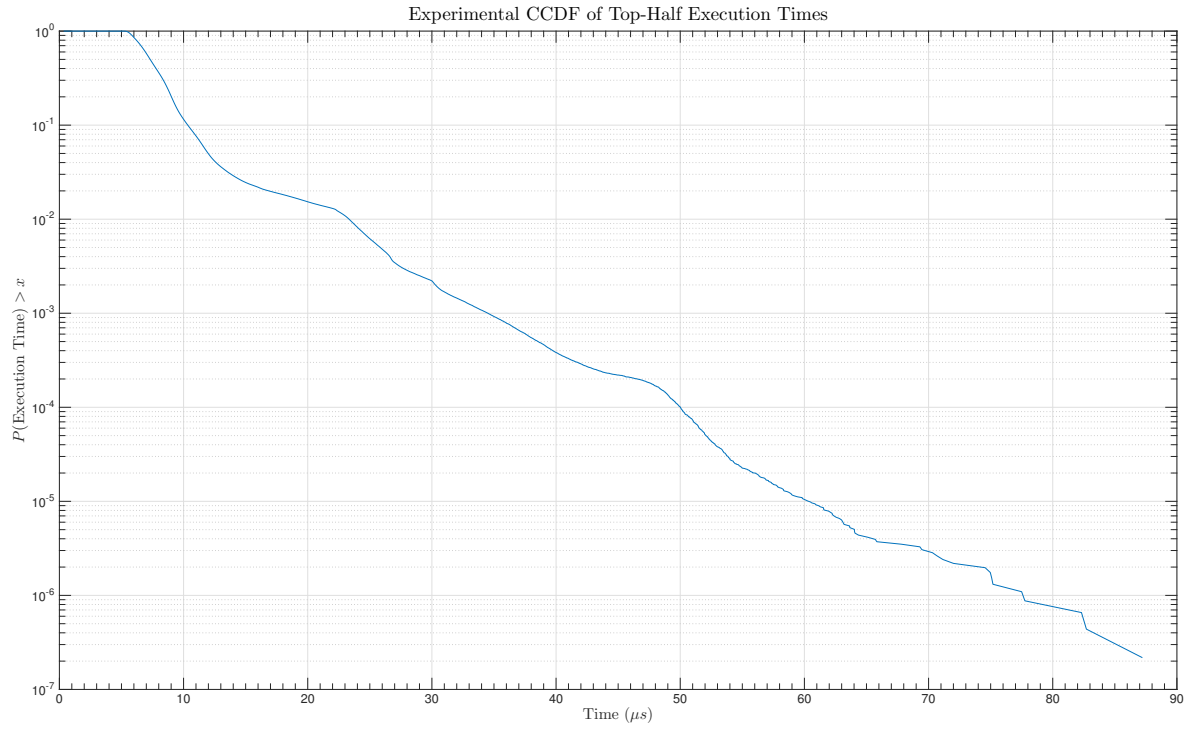
(b) Measurements below the 99<sup>th</sup> percentile.

Figure 4.3: PDF of GPU bottom-half execution time.

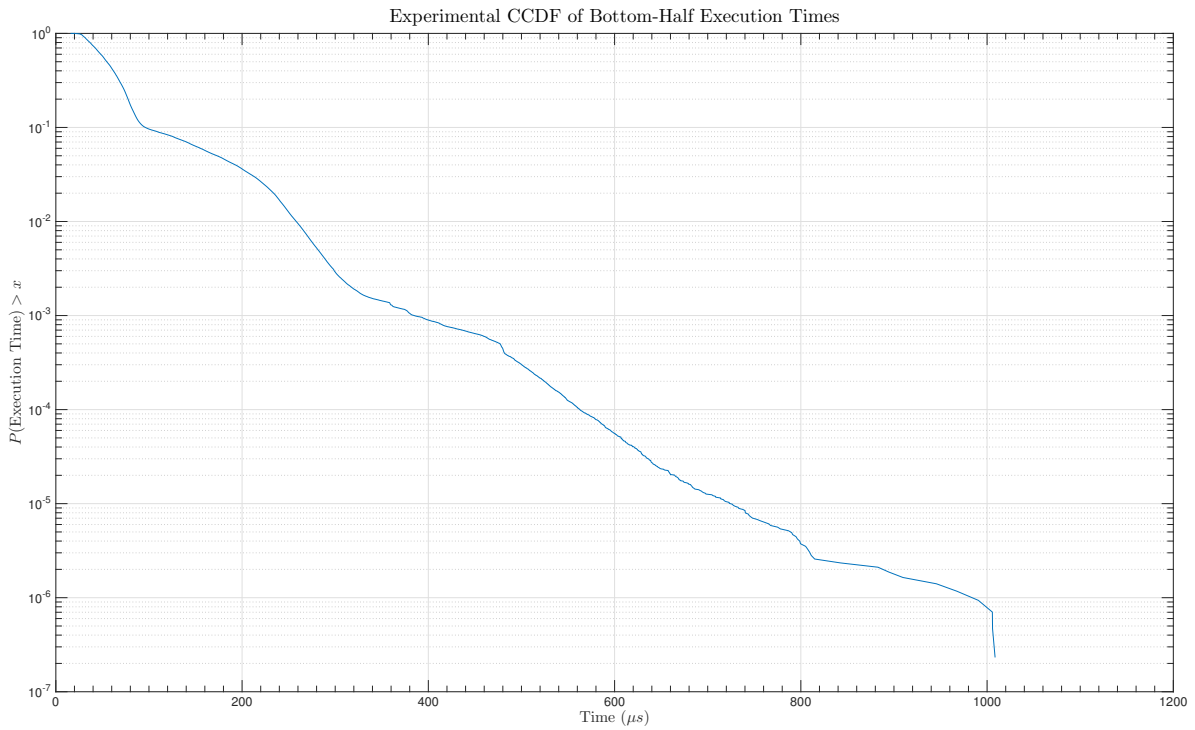
insets (a) and (b), respectively. It is important to note that the y-axis is plotted on a log scale—the log scale makes it easier for us to observe and reason about outlier characteristics. In Figure 4.4(a), we observe that top-half outliers are rare. For instance, only 0.01% ( $y = 10^{-4}$ ) of observed top-halves had execution times greater than about 50 $\mu$ s. Moreover, only 0.001% of observed top-halves had execution times greater than roughly 60 $\mu$ s, and only 0.0001% of observed top-halves had execution times greater than about 78 $\mu$ s.

Bottom-half outliers are also rare. In Figure 4.4(b), we observe that only 0.01% of observed bottom-halves had execution times greater than about 550 $\mu$ s. Moreover, only 0.001% of observed bottom-halves had execution times greater than roughly 750 $\mu$ s, and only 0.0001% of observed bottom-halves had execution times greater than 1000 $\mu$ s.

We conclude our discussion of GPU interrupt handling overheads by stressing the need for including these overheads in schedulability analysis. The CPU time consumed by GPU interrupt handling is not trivial. In our 20-minute experiment, we find that the system spent roughly 36.6 *seconds* executing top-halves. Similarly, about 282.29 seconds were spent executing bottom-halves. Altogether, interrupt processing consumed approximately 2.21% of available CPU time, across twelve CPUs. Moreover, schedulability analysis must be mindful of the very clear discrepancies between worst-case and average measurements.



(a) top-halves



(b) bottom-halves

Figure 4.4: CCDFs of top-half and bottom-half execution times.

#### 4.3.1.2 Memory Overheads

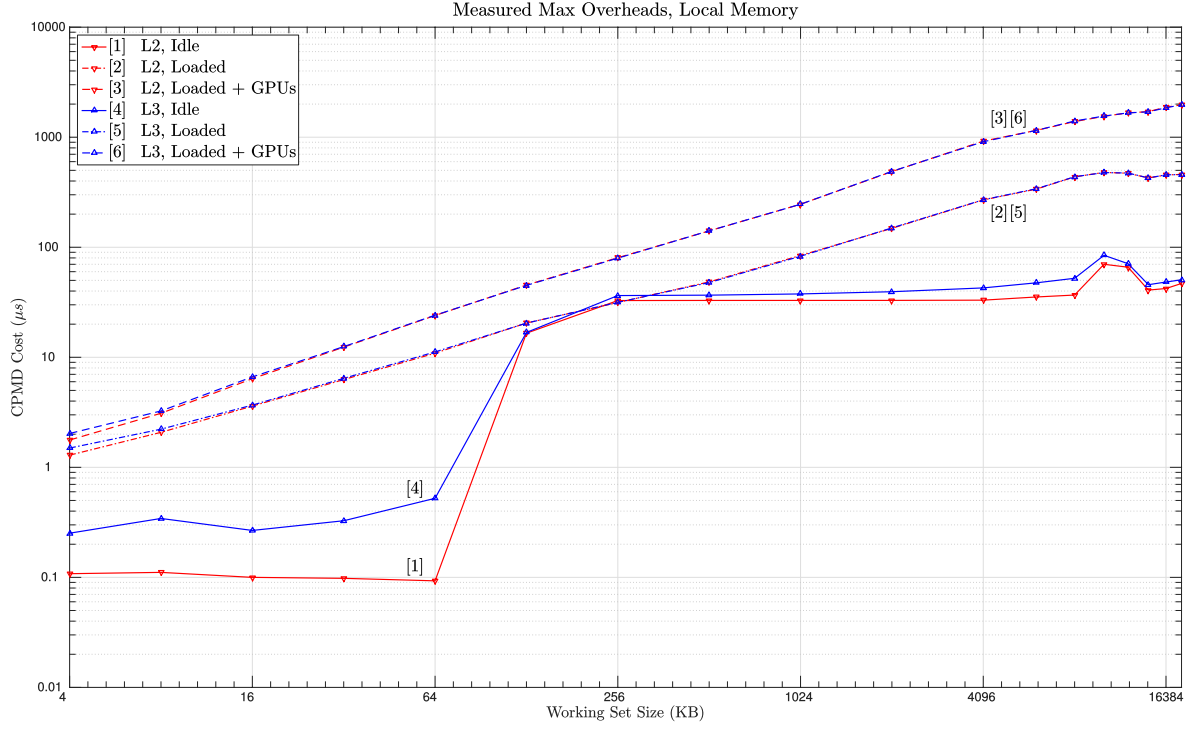
Although algorithmic overheads are important, those related to memory access are more so in a real-time GPU system. As pointed out by Pellizzoni and Caccamo (2010), I/O memory bus traffic can significantly impact the performance of tasks executing on CPUs due to system memory bus contention. Moreover, in multi-GPU platforms, there is also contention for the PCIe bus. We seek to quantify two memory-related overheads. First, we want to determine the impact GPU memory traffic has on CPMDs. Second, we seek to determine the speed at which data can be transmitted to and from system memory and directly between GPUs. We incorporate the former into schedulability analysis. The latter is used to compute task execution requirements on GPU CEs—critical to real-time GPU schedulability.

**The Effect of Bus Contention on CPMDs.** To assess CPMDs, we used an experimental method modeled after the “synthetic method” described by (Brandenburg, 2011b). A non-preemptive instrumented process records the time taken to read a prescribed amount (a “working set size”) of sequential data from a “hot” cache. The process suspends for a short duration, resumes on a random processor, and rereads said data from the now “cold” cache. A cost is determined by subtracting the hot measurement from the cold. On our evaluation platform, individual measurements fall into one of three categories: L2 preemption, L3 migration, and memory migration. An L2 preemption measurement is one where the hot and cold measurements are performed on the *same* CPU, since each CPU has a private L2 cache on our evaluation platform. An L3 migration measurement is one where the hot and cold measurements are performed on CPUs that *share* an L3 cache; on our platform, these are CPUs within the same NUMA node. A memory migration is one where hot and cold measurements are performed on CPUs that do *not share* a cache; on our platform, these CPUs reside within different NUMA nodes.

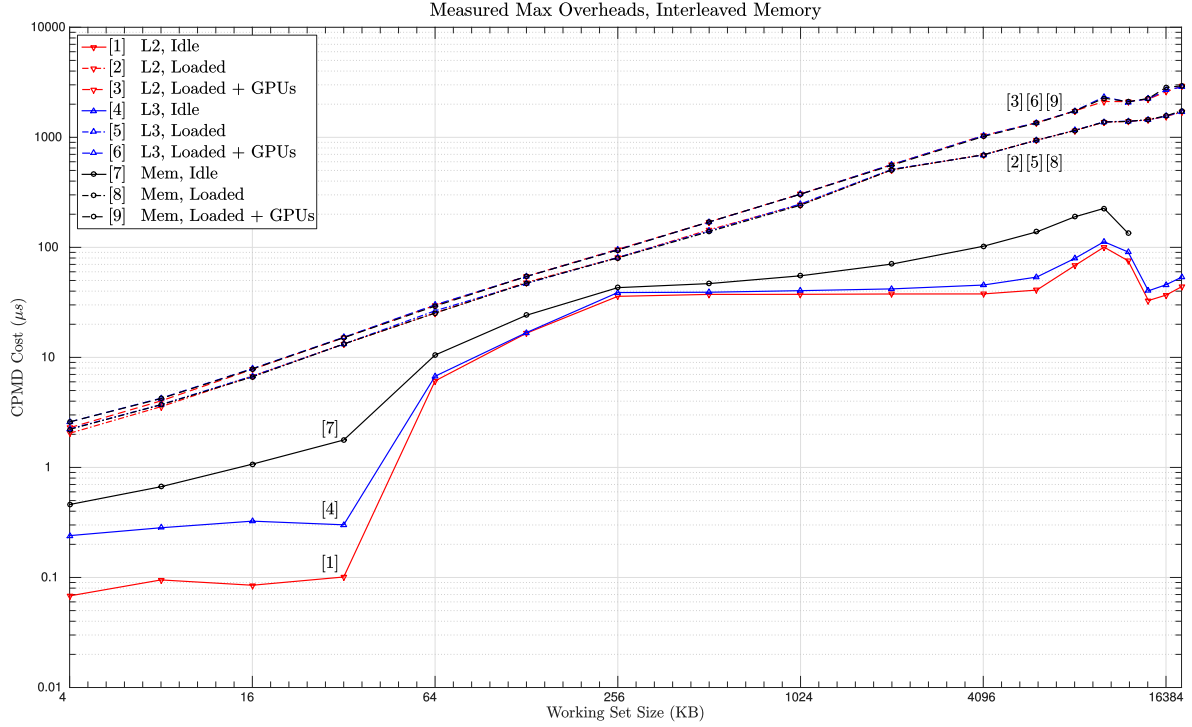
We are concerned with two memory configurations since our test platform is a NUMA platform. Under partitioned and clustered CPU scheduling (when clusters reside entirely within a NUMA node), memory can be allocated locally to increase performance and reduce interference from NUMA-remote tasks. However, under global CPU scheduling, one may *interleave* memory pages across the NUMA nodes in order to obtain good average case performance. We require overhead data for both configurations in order to accurately model each CPU/GPU configuration described in Section 4.3.6.

Under both *local* and *interleaved* configurations, we collected three CPMD datasets: **(i)** An “idle” dataset where the instrumented process runs alone; **(ii)** a “loaded” dataset where “cold” measurements are taken in



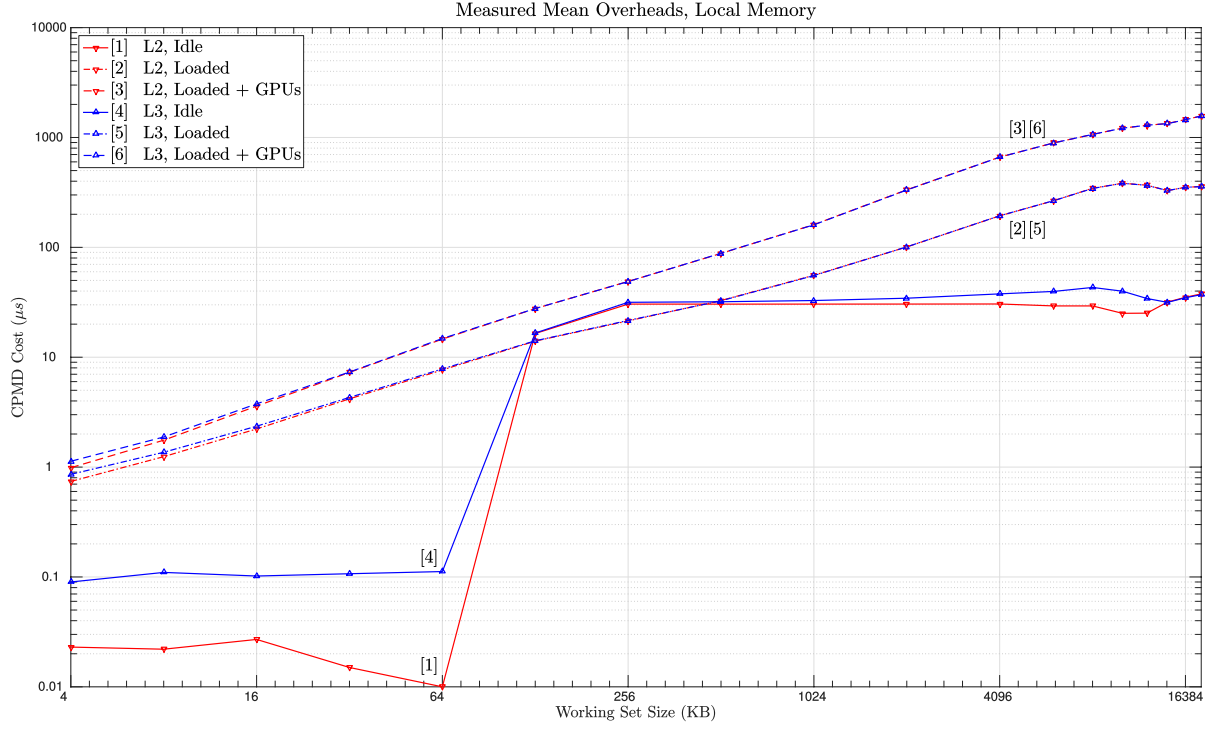


(a) local

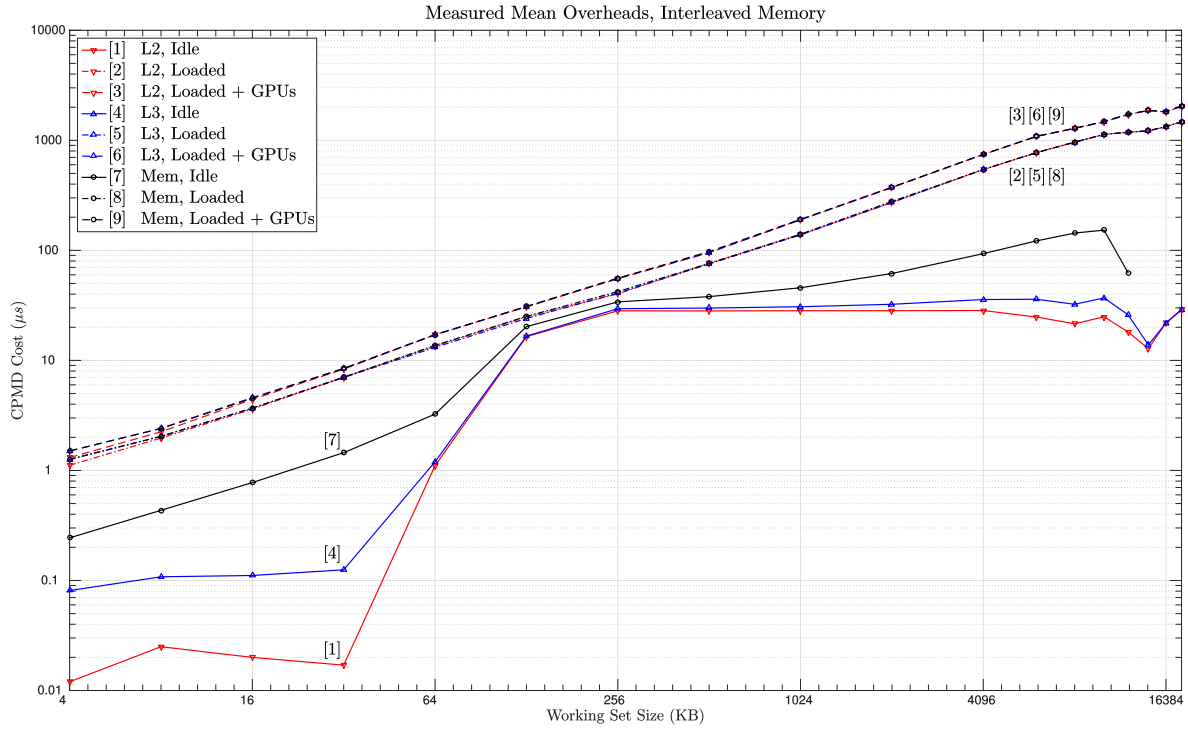


(b) interleaved

Figure 4.5: Considered CPMD maximum overheads due to GPU traffic.



(a) local



(b) interleaved

Figure 4.6: Considered CPMD mean overheads due to GPU traffic.

the presence of cache-trashing processes that introduce contention for both caches and memory bus; and (iii) a “loaded+gpu” dataset where additional load is created by GPU-using processes, one for each CE, that fully loads the bidirectional PCIe bus with a constant stream of 512MB DMA memory transfers to and from pinned pages in system memory. We gathered 5,000 samples apiece to measure L2 preemptions, L3 migrations, and memory migrations for each working set size. We distilled these samples into max and mean values for each type of measurement.<sup>4</sup>

Figure 4.5 plots measured maximum costs for each type of CPMD. Figure 4.5(a) depicts measurements where all accessed memory was local to the NUMA node of the CPU(s). This figure does not include measurements for memory migrations, since there are no memory migrations when data is NUMA-local. Figure 4.5(b) depicts measurements where accessed memory was evenly interleaved, with page granularity, across the two NUMA nodes of our evaluation platform. In both of these figures, the  $x$ -axis uses a  $\log_2$  scale, while the  $y$ -axis uses a  $\log_{10}$  scale. We make two high-level observations from these graphs.

**Observation 1.** *The cache offers little or no benefit in the presence of heavy load.*

In Figure 4.5(b), observe that the curves for each type of measurement within the loaded data sets practically coincide for working set sizes of 16KB or greater. For example, curves 2, 5, and 8 are virtually indistinguishable. We can make similar observations in Figure 4.5(a).

**Observation 2.** *CPMDs on an idle platform are characterized by two plateaus, with an abrupt increase around working set sizes of 128KB for CPMDs for local memory access, and working set sizes of 64KB for CPMDs for interleaved memory access.*

In Figure 4.5(a), find the curves for L2 preemptions (curve 1) and L3 migrations (curve 4). There is very little variation among these CPMDs for working set sizes of 4KB, 8KB, 16KB, 32KB, and 64KB. Also, these CPMDs are on the order of hundreds of nanoseconds—we expect CPMDs to be small because cache reuse should be high in an idle platform. CPMDs increase abruptly for working set sizes of 128KB. However, the increases level off for working set sizes of 256KB and greater. The increase in CPMDs for working set sizes of 128KB is explained by L2 cache the utilization. Although a 256KB L2 cache is only half-filled by a working set size of 128KB, this L2 may still hold other data such as program instructions, OS memory, and

---

<sup>4</sup>We distribute the measurement tools we developed to assess memory overheads as open source under the GNU General Public License, version 2. The source code for these tools is currently available at [www.github.com/GElliott](http://www.github.com/GElliott). Portions of this code is derived from that developed by Brandenburg (2011b).

the memory of other processes. The plateau for working set sizes no less than 256KB is explained by the inclusive 12MB L3 cache, which is large enough to hold the bulk of the tested working sets (also, the relative share of the L3 cache consumed by program code and the memory of the OS and other processes decreases with the larger cache). It is not surprising that CPMD costs begin to increase for working set sizes larger than 12MB. We can make similar observations for CPMDs for interleaved memory access in Figure 4.5(b). However, the first significant increase in CPMDs (curves 1, 4, and 7) occurs at working set sizes of 32KB.

Figure 4.6 plots measured *mean* costs for each type of CPMD. We see the same trends in Figure 4.6 that we see in Figure 4.5.

For a deeper study of CPMDs, we direct the reader to the work of Brandenburg (2011b), which provides a more in-depth investigation of CPMDs. We note that the general trends in Figures 4.5 and 4.6 are consistent with those reported by Brandenburg. More relevant to the topics covered by this dissertation is the *increase* in CPMDs due to GPU memory traffic. We investigate this next.

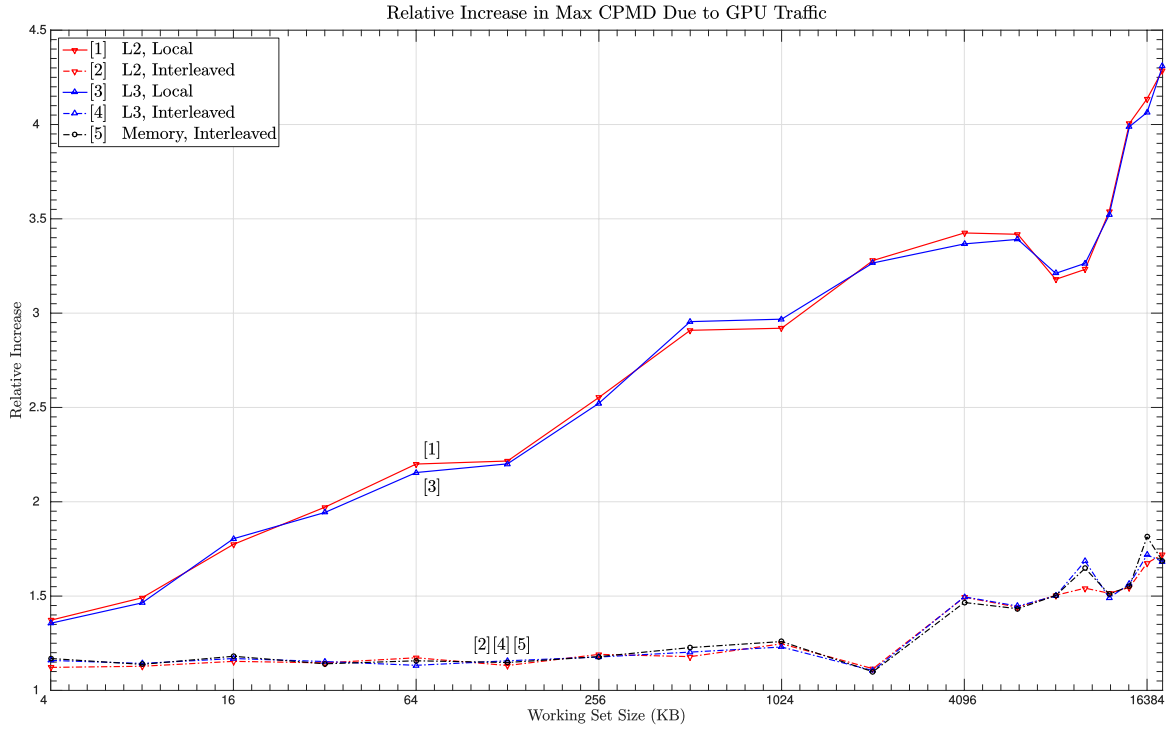
Figure 4.7 plots the relative increase in CPMD costs of select working set size “loaded+gpu” datasets with respect to the “loaded” data set. Figure 4.7(a) relates these increases in terms of maximum CPMD costs, while Figure 4.7(b) relates these increases in terms of mean CPMD costs. We make two observations.

**Observation 3.** *Maximum and mean CPMDs are affected similarly by GPU traffic.*

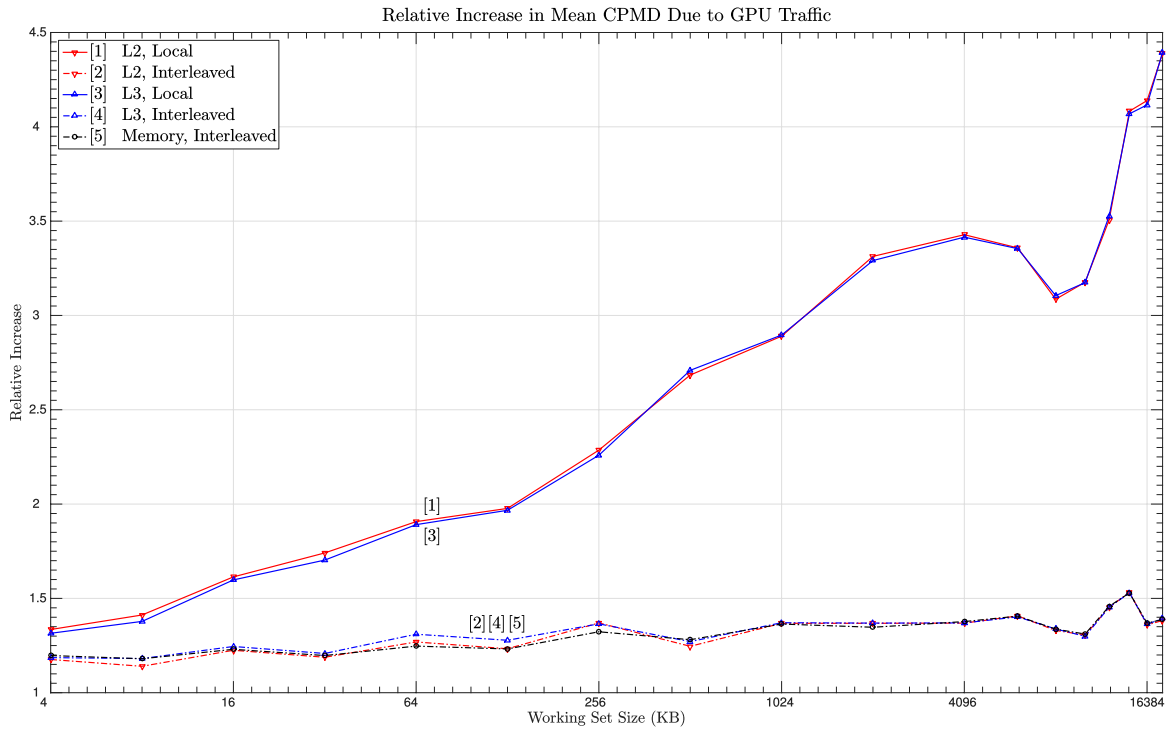
The shape and magnitude of the corresponding curves in Figures 4.7(a) and 4.7(b) are very similar. For example, the plots for L2 preemption CPMDs (curve 1) both exhibit a dip in costs for working set sizes around 8192KB.

**Observation 4.** *GPU traffic affects CPMDs for local memory access more strongly than CPMDs for interleaved memory access: CPMDs for local memory access increase by factors between two and four, while CPMDs for interleaved memory access increase by factors between one and two.*

In Figures 4.7(a) and 4.7(b), we observe that GPU traffic increased CPMDs for local memory access by a factor between two and four for working set sizes larger than 32KB (curves 1 and 3). CPMDs for interleaved memory access were affected to a lesser degree, with increases by a factor between approximately 1.1 and 1.9. However, CPMDs for interleaved memory access *without* GPU traffic are nearly as great as local CPMDs *with* GPU traffic. For example, in Figure 4.6(a), find the L2 preemption CPMD cost for a platform under load with GPUs (curve 3) for a working set size of 4096KB—it is roughly 650 $\mu$ s. Compare this to



(a) maximum



(b) mean

Figure 4.7: Increase in considered CPMD overheads due to GPU traffic.

the L2 preemption CPMD cost for a platform under load *without* GPUs (curve 2) for interleaved memory in Figure 4.6(b)—it is about 550 $\mu$ s.

The above measurements demonstrate that GPU traffic must be considered when deriving estimates of CPMD overhead. Moreover, these considerations must be cognizant of memory locality on a NUMA platform.

**The Effect of Bus Contention on GPU DMA Costs.** Under load, GPU DMAs experience contention for the following buses: the GPU-internal memory bus, several PCIe buses at various hierarchical levels), the processor-I/O hub interconnect, and the system memory bus. If memory is interleaved across NUMA nodes, then additional contention can be experienced for the processor-processor (NUMA) interconnect as well as the remote system memory bus.

We performed experiments to determine GPU DMA costs using a technique similar to the one we used to determine CPMD overheads. An instrumented process performed DMA operations to and from system memory and peer-to-peer DMA operations between GPUs. We tested both local and interleaved configurations under idle and loaded scenarios. In addition to loading every GPU CE, we also executed *memory-heavy GPU kernels* on the EE of GPUs used by the instrumented process in order to stress the GPU's own local memory bus. We took 50,000 measurements for each type of DMA operation for each tested DMA operation size.

Figure 4.8 shows the maximum and mean DMA times for local and interleaved memory access that we observed on idle platform.<sup>5</sup> We make two observations.

**Observation 5.** *On an idle platform, all DMA operation types have similar performance curves: for smaller DMA operations, setup costs are dominant; for larger DMA operations, the costs to transmitting data are dominant.*

In each of the four insets of Figure 4.8, we observe that all curves are similar. Overhead costs to setting up DMA operations dominate for smaller DMA operations (those no greater than 64KB)—this is indicated by the relatively flat portion the curves that connect the cost data-points of smaller DMA operation. The cost of DMA operations begin to scale linearly with DMA operations size for larger DMA operations (those

---

<sup>5</sup>Although peer-to-peer DMA operations should not be affected by interleaved memory access of *system memory* in an idle platform, we include the costs of these operations in order to allow direct comparison against the costs of GPU-to-system and system-to-GPU DMA operations.

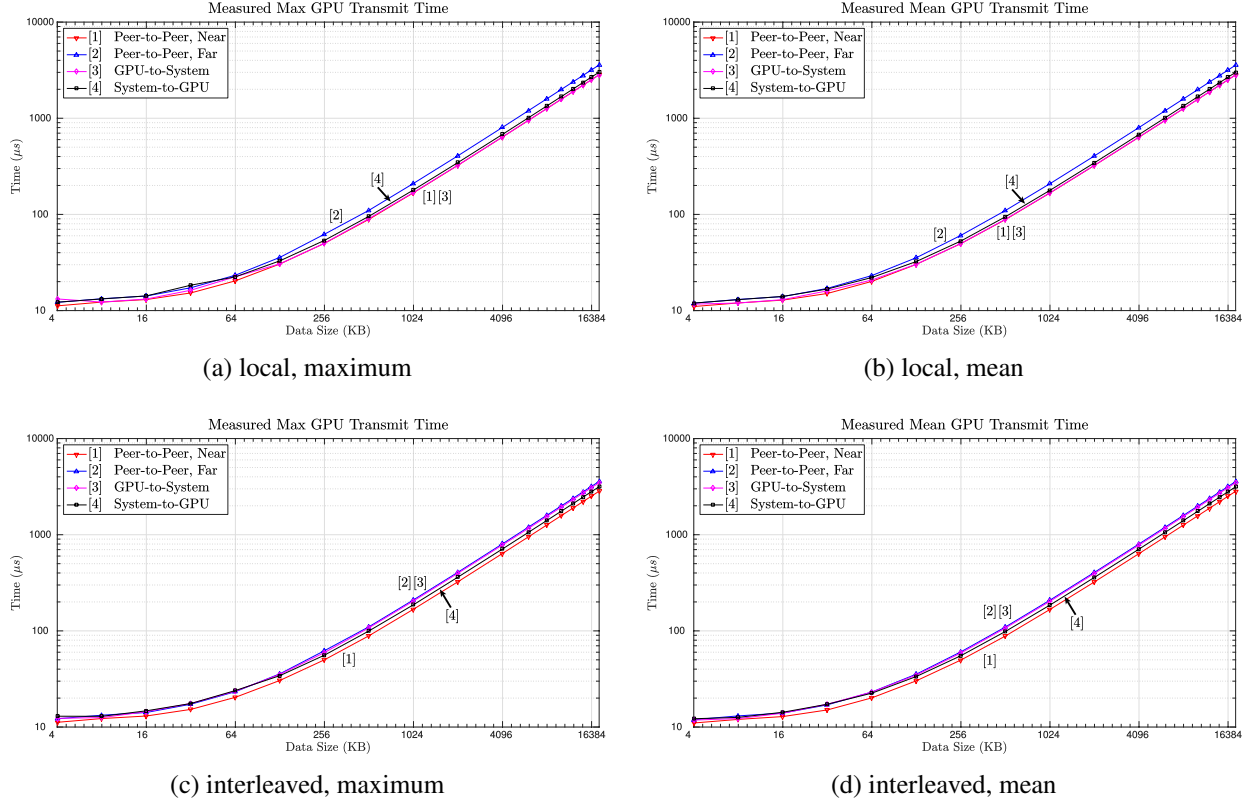


Figure 4.8: GPU data transmission time in an idle system.

no less than 128KB). For example, in Figure 4.8(a), observe that far peer-to-peer DMA (curve 2) takes approximately 200 $\mu$ s for 1024KB, 400 $\mu$ s for 2048KB, and 800 $\mu$ s for 4096KB—cost doubles as DMA size doubles. The curves for the other DMA operation types share the same slope as the one for far peer-to-peer DMA, indicating the same trend. The data reflected by Figure 4.8 is very consistent with those reported by Kato *et al.* (2011a) and Fujii *et al.* (2013).

**Observation 6.** *On an idle platform, far peer-to-peer DMA operations may be more costly than DMA operations to or from system memory.*

In Figures 4.8(a) and 4.8(b) (local memory DMA), observe that the curve for far peer-to-peer DMA operations (curve 2) lies *above* the other curves for DMA operations no less than 32KB. In Figures 4.8(a) and 4.8(b) (interleaved memory DMA), the curve for GPU-to-system memory (curve 3) practically coincides with that of far peer-to-peer DMA. The relative high cost of far peer-to-peer DMA is surprising, since data only traverses the PCIe bus in peer-to-peer DMA operations. We may rule out costs due to coordination between the peer GPUs, since near peer-to-peer DMA operations would be equally costly if this were the

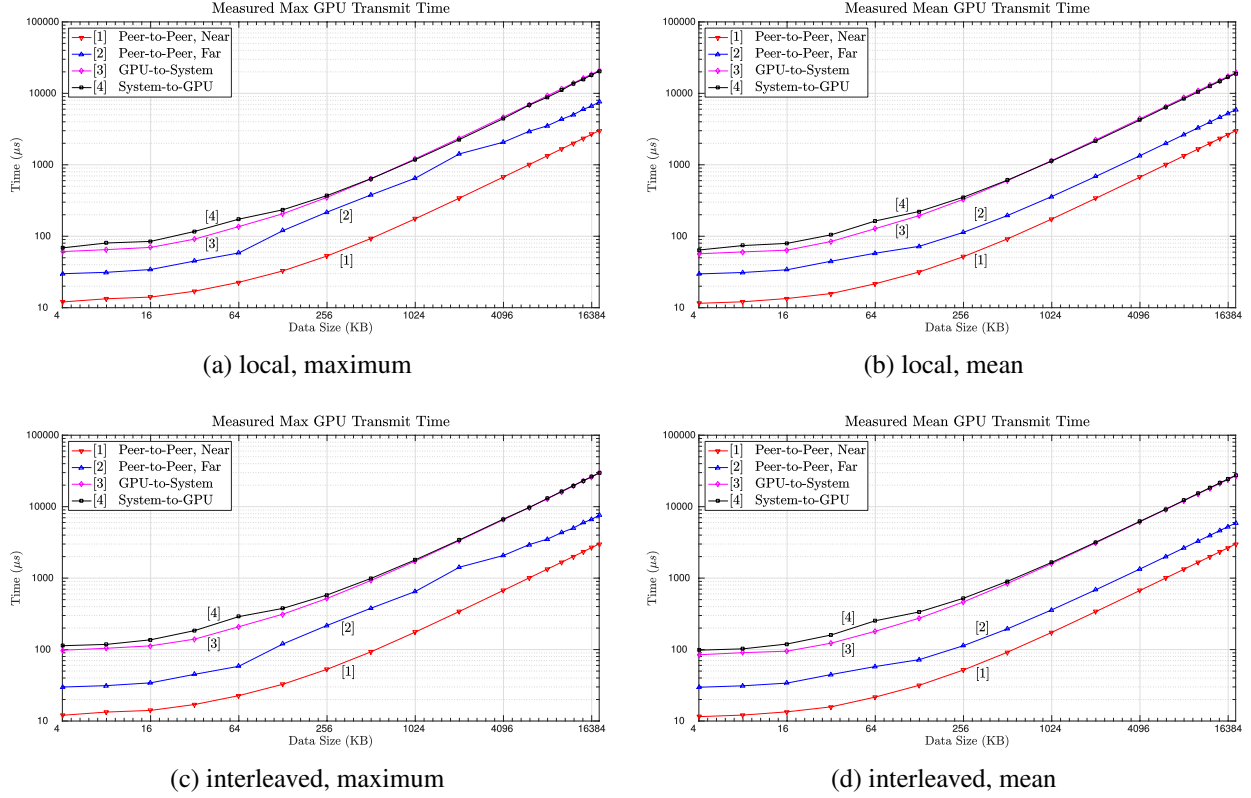


Figure 4.9: GPU data transmission time in system under load.

case. We can only speculate that the PCIe root complex that connects far GPUs may not be as efficient as the PCIe switch that connects near GPUs. Nevertheless, the use of far peer-to-peer DMA may still be more efficient for transmitting data between two GPUs than using two DMA operations to bounce the data through system memory.

We now examine DMA operation costs when our evaluation platform is under load. Figure 4.9 shows the maximum and mean DMA costs for local and interleaved memory access when the platform is under load. We draw only high-level observations from Figure 4.9, deferring observations of comparative performance against DMA costs in an idle platform to another set of figures.

**Observation 7.** *System-to-GPU DMA is more costly than GPU-to-system DMA; GPU-to-system DMA is more costly than far peer-to-peer DMA; far peer-to-peer DMA is more costly than near peer-to-peer DMA.*

We may observe this in every inset of Figure 4.9. The DMA cost curve for system-to-GPU DMA (curve 4) lies above the curve for GPU-to-system DMA (curve 3), even though the curves are relatively close (especially for DMA operations no less than 1024KB). The curve for GPU-to-system DMA clearly lies above



the curve for far peer-to-peer DMA (curve 2). Similarly, the curve for far peer-to-peer DMA lies above the curve for near peer-to-peer DMA (curve 1).

**Observation 8.** *For DMA operations no less than 256KB, the mean cost of far peer-to-peer DMA is approximately twice as much the mean cost of near peer-to-peer DMA.*

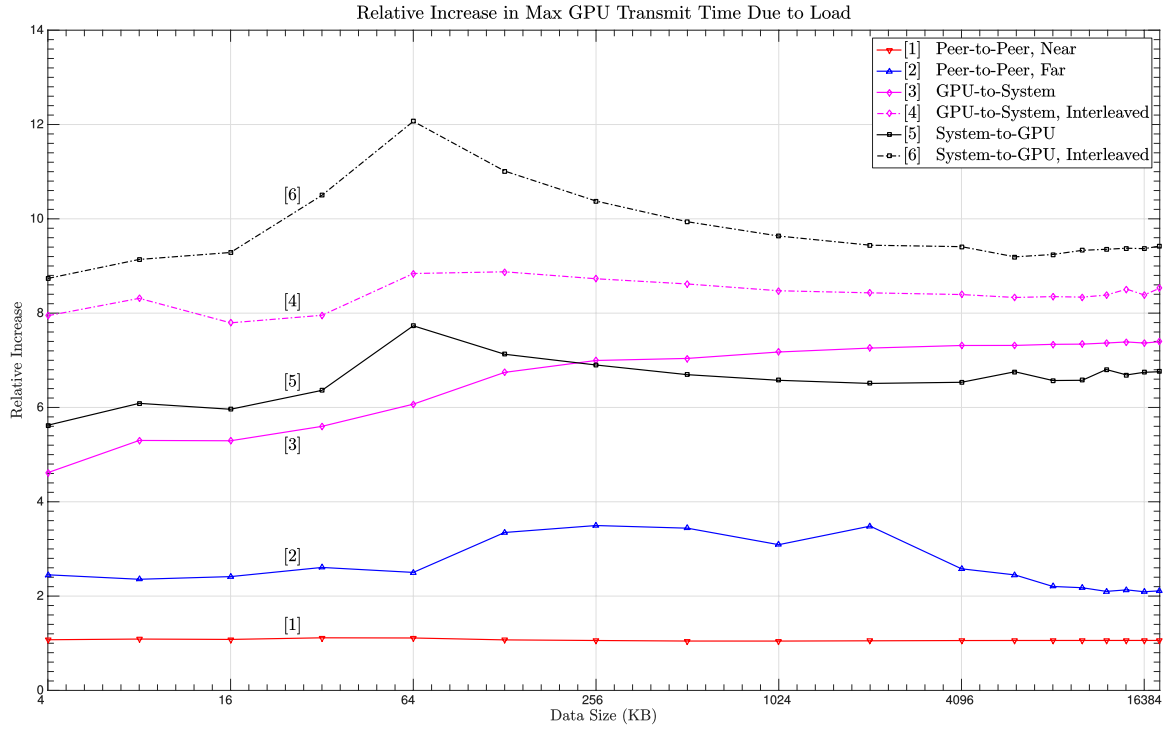
We can make this observation in Figure 4.9(b). Find the value of curve 1 for near peer-to-peer for DMA sizes of 256KB—it is approximately 50 $\mu$ s. Find the value of curve 2 for far peer-to-peer DMA for the same DMA size—it is approximately 110 $\mu$ s, a little more than twice the cost for near peer-to-peer DMA. We expect this behavior from the PCIe bus topology of our evaluation platform. Under load, far peer-to-peer DMA operations receive half as much PCIe bandwidth, on average, as near peer-to-peer DMA operations. This is because far peer-to-peer DMA operations must traverse an additional (loaded) PCIe switch, which reduces the available bandwidth to the DMA operation by half.

We now investigate the effect load and page interleaving has on DMA costs more directly. Figure 4.10 depicts observed *increases* in maximum and mean DMA costs due to load. We observe the following.

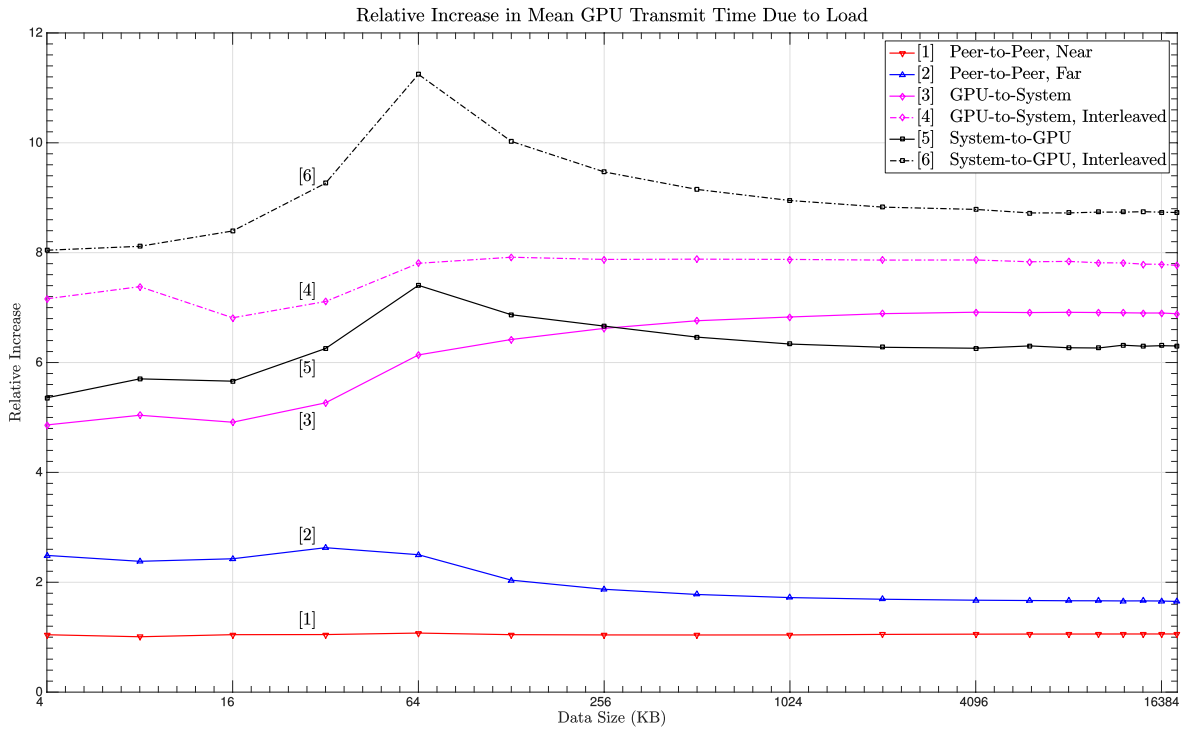
**Observation 9.** *Load can cause significant increases in DMA costs, so it must be considered in schedulability analysis.*

Consider the case where four GPUs share a PCIe bus, as they do in Figure 2.20. Under load, one might assume that each GPU will obtain 25% of the PCIe bus’s bandwidth—increasing DMA costs by a factor of four. However, such an assumption ignores the effect of contention for the system memory bus. We see in Figure 4.10 these cost increases can be considerably greater. For example, consider curves for GPU-to-system and system-to-GPU DMA (curves 3 and 5, respectively) in both Figure 4.10(a) and Figure 4.10(b). Here, we see that DMA costs generally increase by factors between five to eight times. DMA cost increases are even greater when memory pages accessed by GPUs are interleaved across the NUMA nodes. To observe this, find the curves for GPU-to-system and system-to-GPU DMA with interleaved system memory (curves 4 and 6, respectively) in both Figure 4.10(a) and Figure 4.10(b). We see that the increase in DMA costs generally increase between eight to ten times, but may still be as great as twelve times (curve 6, for data sizes of 64KB, in Figure 4.10(a)).

*Ultimately, this result shows us that bus contention must be accounted for in schedulability analysis. We consider this a significant oversight of prior work in real-time GPU research.*



(a) maximum



(b) mean

Figure 4.10: Increase in DMA operation costs due to load.

**Observation 10.** *Near peer-to-peer DMA operations are hardly affected by load. Far peer-to-peer operations are moderately affected.*

Despite added contention for the GPU’s *local* memory bus caused by the memory-heavy GPU kernel executing on the EE of the tested GPU, we observe that load hardly affects near peer-to-peer DMA: curve 1 in both insets of Figure 4.10 are very close to 1.0 (never exceeding a factor of 1.1) for all tested data sizes. Far peer-to-peer memory copies exhibit an increase factor between 2.1 to 3.5 for maximum DMA costs and an increase factor between 1.7 to 2.6 for mean DMA costs, as respectively reflected in Figure 4.10(a) and Figure 4.10(b) by curve 2. From this observation, we learn that costs due to contention for the PCIe bus are not negligible.

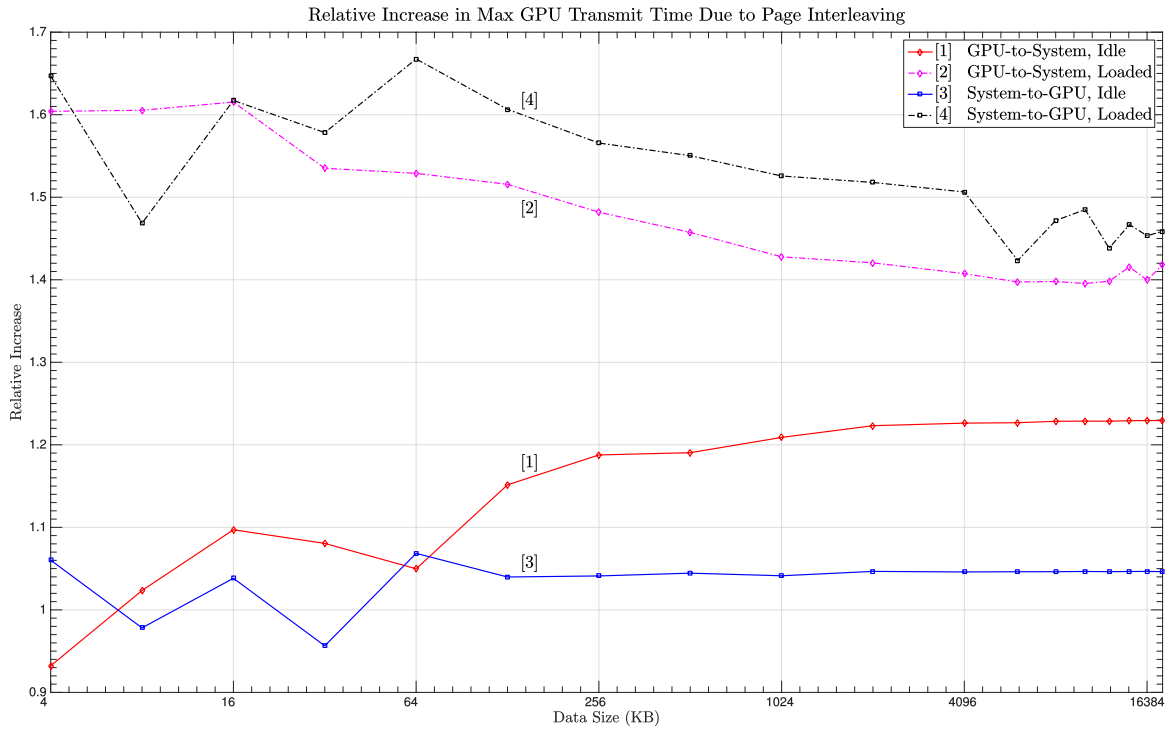
Figure 4.11 depicts observed *increases* in maximum and mean DMA costs caused by interleaving memory pages across NUMA nodes in system memory. The insets in this figure do not include curves for peer-to-peer DMA, since these operations do not touch interleaved pages in system memory.

**Observation 11.** *In general, GPU DMA performance is not improved by page interleaving.*

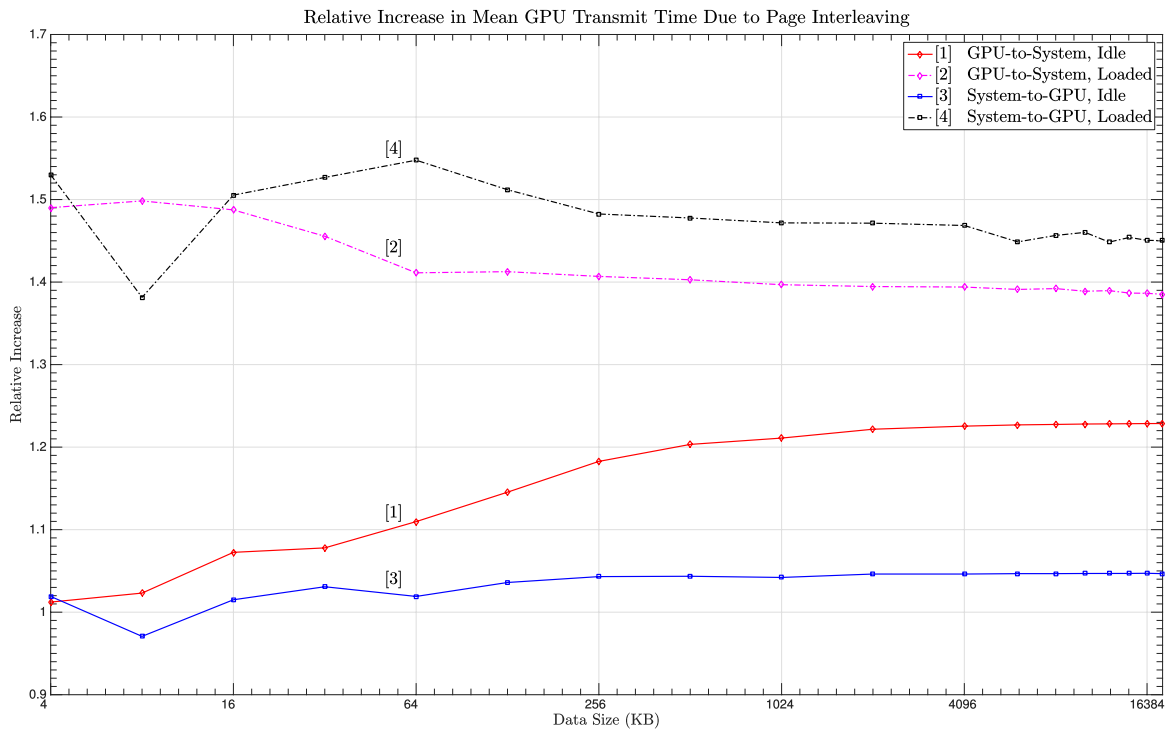
One may suspect that interleaving pages among NUMA nodes may actually *improve* GPU DMA performance, since memory accesses in such a scenario may operate in parallel. However, in general, interleaving pages across NUMA nodes usually increases GPU DMA costs, even when the platform is idle. We see that nearly every data point for the curves in Figure 4.11(a) and Figure 4.11(b) lie above 1.0, indicating increases in DMA cost. For example, maximum GPU-to-system DMA costs increased by roughly 20% and 40% for data sizes no less than 256KB on idle (curve 1) and loaded (curve 2) platforms, respectively, as depicted in Figure 4.11(a). There are some exceptions where page interleaving may lead to decreased DMA costs. For instance, in Figure 4.11(a), page interleaving reduced maximum DMA costs for some DMA data sizes no greater than 32KB on an idle platform, as indicated by curves 1 and 3. However, practically speaking, it is unlikely that DMA operations will always be performed on a completely idle platform. We claim that meaningful performance benefits from page interleaving cannot be achieved on our evaluation platform.

**Observation 12.** *Page interleaving has a stronger effect on DMA costs on a platform under load.*

As we discussed above, interleaving may introduce additional bus contention, especially for the bus connecting the two NUMA nodes. Here, from curves 2 and 4 in Figure 4.11(a) and Figure 4.11(b), we see that the cross-traffic between NUMA nodes caused by interleaving results in increased DMA costs—these



(a) maximum



(b) mean

Figure 4.11: Increase in DMA operation costs due to page interleaving.

curves always lie above those for an idle platform. Page interleaving increased maximum DMA operation costs in a loaded platform between 40% and 65% (Figure 4.11(a)). The increase in mean DMA operation costs fell within nearly the same range: between about 40% and 55% (Figure 4.11(b)).

This concludes our investigation of system overheads introduced by GPUs. Next, we discuss how we incorporate these overheads into overhead-aware schedulability analysis.

### 4.3.2 Scope

GPUSync is highly configurable and adaptable. It may be used with any JLFP scheduler. It supports a mix of CPU and GPU cluster configurations. GPUSync can support GPUs with zero, one, or two CEs.<sup>6</sup> Also, the number of tokens per GPU and maximum FIFO length are configurable. GPU migration may be carried out through peer-to-peer DMA or through a temporary buffer in system memory. In support of peer-to-peer migration, CE locks may be requested one at a time, or once as a DGL. Engine locks may be configured to satisfy pending requests in FIFO- or priority-order. Finally, any of the four GPU Allocator heuristics we employ may be disabled without breaking real-time predictability. There are well over 100,000 different GPUSync configurations that are worthy of study.

We must limit the scope of the configurations we examine in order to make our study tractable. This scope must be focused enough so that we are not overwhelmed with data, and yet it must be broad enough so that we may come to understand the tradeoffs among general classes of GPUSync configurations. For the sake of presentation, we define our scope in its entirety here in one place. Our configuration choices are as follows.

1. **Twelve CPU/GPU configurations.** We wish to understand the tradeoffs in schedulability among the combinations of CPU and GPU cluster configurations, as we described in Chapter 1. As a consequence, we must study every reasonable combination of CPU and GPU cluster configurations. Nine of the twelve configurations we study are depicted in Figure 4.1. The remaining three configurations are those where GPUs are put in larger clusters of four, rather than clusters of two. This enables us to determine what effect far peer-to-peer migration costs may have on schedulability.

---

<sup>6</sup>In Section 2.3, we discussed integrated GPUs, which lack CEs. Although we do not focus on integrated GPUs in this dissertation, we discuss GPUSync's support for such GPUs in Chapter 6.

2. **Peer-to-peer and system memory migration.** As we discussed in Section 4.3.1.2, peer-to-peer DMA has the potential to greatly reduce GPU migration costs, since data is handled less often and peer-to-peer DMA can be significantly faster than DMAs that involve system memory. However, peer-to-peer DMA requires a migrating task to hold two CE locks simultaneously. As we will see in Sections 4.3.4.2 and 4.3.4.3, this has a significant impact on pi-blocking analysis. Studying both peer-to-peer and system memory migration methods allows us to determine if peer-to-peer DMA is efficient enough to overcome more pessimistic pi-blocking bounds.
3. **Optimal configurations of the GPU Allocator with the number of tokens per GPU defined as  $\rho \in \{1, 2, 3\}$ , and a limited set of non-optimal GPU Allocator configurations for  $\rho = \infty$ .** We study a wide selection of values for  $\rho$  in order to determine if GPU engine parallelism can improve schedulability. We primarily focus on (suspension-oblivious) optimal configurations of the GPU Allocator where  $\rho \in \{1, 2, 3\}$ . This gives us a spectrum of token values to study. Exclusive GPU allocation is represented by  $\rho = 1$ , while the potential for full GPU engine utilization is represented by  $\rho = 3$ . For partitioned GPU configurations, we also study the effect of essentially eliminating the token lock. Here, we render all token lock request trivial by setting  $\rho = \infty$ .
4. **FL scheduling.** We limit our study to FL-based schedulers. This may seem like an odd choice, given our stated motivation for supporting automotive applications. EDF-like global and clustered schedulers, such as C-FL, are commonly associated with soft real-time systems, *i.e.*, those generally regarded as non-safety-critical. This is partly due to generally weaker job response-time guarantees. However, in an automotive setting, the reaction time of an *alert* driver is about 700ms (Green, 2000). Such a reaction time is well within the realm of possibility under bounded deadline tardiness constraints. Moreover, EDF-like schedulers come with the added benefit of the absence of severe utilization constraints. We choose to investigate schedulability under FL-based schedulers because these schedulers have the best known bounds on deadline tardiness.
5. **FIFO-ordered engine locks.** We limit our scope to FIFO-ordered engine locks because we expect that they will yield better schedulability under FL scheduling than priority-ordered locks. Generally speaking, FIFO-ordered locks result in less pessimistic bounds on pi-blocking for deadline-based schedulers, since the analysis for priority-ordered locks under deadline-based schedulers must generally assume that each issued request has the lowest priority for long durations of time.

6. **Use of DGLs.** DGLs limit the effect of transitive blocking under nested locking. For peer-to-peer migrations, we assume that CE locks are acquired through atomic DGL requests, since this will give us better schedulability results.
7. **Two CEs per GPU.** We focus our attention on GPUs with two CEs. We make this decision because GPUs with two CEs provide us with a richer platform for schedulability studies. However, we recognize the importance of GPUs with a single CE. For this reason, we present detailed blocking analysis for such GPUs in Section 4.3.4.3. However, our schedulability experiments assume dual-CE GPUs.
8. **1MB DMA chunk size.** In our schedulability experiments, we assume that each type of DMA operation (*i.e.*, input, output, and state data) are broken into an integral number of 1MB chunks, plus at most one fractional chunk. We select this chunk size because it is large enough that it is not dominated by setup cost overheads, while keeping DMA operation sizes small enough to prevent long durations of CE blocking.

Choices 1, 2, and 3 give us a broad selection of GPUSync configurations to study. The remaining choices help keep our study tractable, and are also meant to maximize real-time schedulability under GPUSync for the systems we are motivated to study.

### 4.3.3 Task Model for GPU-using Sporadic Tasks

We require a task model that adequately describes a task set of GPU-using tasks. We extend the sporadic task model described in Section 2.1.1 with additional notation.

We consider a task system,  $\mathcal{T}$ , comprised of  $n$  real-time tasks  $T_1, \dots, T_n$  that are scheduled on  $m$  CPUs, partitioned into clusters of  $c$  CPUs each. The subset  $\mathcal{T}^{gpu} \subseteq \mathcal{T}$  includes all tasks that require GPU resources from the system's  $h$  GPUs, partitioned into clusters of  $g$  GPUs each. The subset  $\mathcal{T}^{cpu} \triangleq \mathcal{T} \setminus \mathcal{T}^{gpu}$  are tasks that do not use a GPU. The tasks in  $\mathcal{T}$  are partitioned among the CPU and GPU clusters. We denote the set of tasks assigned to the  $a^{th}$  GPU cluster by  $\mathcal{T}_a^{gpu}$ , where  $a$  is a GPU cluster index, which starts from zero. We similarly denote the set of tasks assigned to the  $a^{th}$  CPU cluster by  $\mathcal{T}_a$ . The parameter  $q_i$  denotes  $T_i$ 's *provisioned GPU execution time* on an execution engine. The parameter  $e_i^{gpu}$  denotes  $T_i$ 's total CPU execution time requirements *within* its GPU critical section (note that we assume  $e_i^{gpu}$  is included in  $e_i$ ). Each job  $J_{i,j}$  sends  $z_i^I$  bytes of data as *input* to GPU computations. Similarly, each job  $J_{i,j}$  receives  $z_i^O$  bytes of data as *output* from GPU computations. The size of job  $J_{i,j}$ 's state that may be migrated among GPUs is denoted

by  $z_i^S$ . For convenience, we define the function  $xmit(z_i^I, z_i^O, z_i^S)$  to specify the total data transmission time required by job  $J_{i,j}$ . The value of this function can be computed given the empirical measurements described in Section 4.3.1.2. We assume that a job of  $T_i \in \mathcal{T}^{gpu}$  may use any one arbitrary GPU in its GPU cluster. For  $T_i \in \mathcal{T}^{cpu}$ , the parameters  $q_i$ ,  $z_i^I$ ,  $z_i^O$ , and  $z_i^S$  are zero. Finally, we redefine task utilization to incorporate GPU execution time, such that

$$u_i \triangleq \frac{e_i + q_i + xmit(z_i^I, z_i^O, z_i^S)}{p_i}. \quad (4.1)$$

We define several additional terms for the purpose of locking analysis. Let  $L_i^K$  denote the maximum token critical section length of task  $T_i$ ,  $b_i^K$  denote the maximum time job  $J_{i,j}$  may be blocked due to the token lock, and  $b_i^E$  denote the maximum time  $J_{i,j}$  may be blocked *within* a token critical section for *all* engine locks. Let  $Z$  denote a configured DMA chunk size that is used to break large DMA operations into smaller ones. We denote the number of chunks required to transmit task data by:  $N_i^I \triangleq \lceil z_i^I / Z \rceil$ ;  $N_i^O \triangleq \lceil z_i^O / Z \rceil$ ;  $N_i^S \triangleq \lceil z_i^S / Z \rceil$ . Let  $X^I$ ,  $X^O$ , and  $X^{P2P}$  denote the maximum time it takes to transmit a chunk of GPU data for input, output, and peer-to-peer migration, respectively, and let  $X^{max}$  denote the maximum of  $X^I$ ,  $X^O$ , and  $X^{P2P}$ . Finally, let  $S_i$  denote the maximum time to perform a GPU migration. For peer-to-peer migrations,

$$S_i \triangleq X^{P2P} \cdot N_i^S. \quad (4.2)$$

For migrations through system memory,

$$S_i \triangleq (X^I + X^O) \cdot N_i^S. \quad (4.3)$$

For platform configurations with partitioned GPUs,  $S_i = 0$ .

Table 4.1 summarizes these terms, as well as additional terms that we use later in analysis. In the subsequent sections, we will refer to the terms listed in Tables 2.1, 2.2, and 4.1.

#### 4.3.4 Blocking Analysis

We now discuss the method we use to bound the length of time a job may be pi-blocked due to GPU token and engine lock requests. We begin by outlining our three-phase process to computing these bounds. We



Additional Task Set and Scheduler Parameters	
$h$	number of GPUs
$g$	GPU cluster size
$\mathcal{T}^{gpu}$	set of GPU-using tasks in the task set $\mathcal{T}$
$\mathcal{T}^{cpu}$	set of CPU-only tasks in the task set $\mathcal{T}$
$\mathcal{T}_a^{gpu}$	set of GPU-using tasks in the task set $\mathcal{T}^{gpu}$ that execute on the $a^{\text{th}}$ GPU cluster
$\widehat{\mathcal{T}}_a$	set of tasks scheduled on CPU clusters associated with the $a^{\text{th}}$ GPU cluster
$Z$	DMA chunk size
Additional Parameters of Task $T_i$	
$q_i$	job worst-case execution time on an GPU execution engine
$e_i^{gpu}$	portion CPU execution time of a task within its token critical section
$z_i^I$	bytes sent to GPU as kernel input
$z_i^O$	bytes received from GPU as kernel output
$z_i^S$	bytes of state data that resides on a GPU
Additional Blocking Analysis Parameters	
$L_i^K$	token critical section length
$b_i^K$	token request blocking bound
$v_i$	number of token critical sections that may block job $J_i$
$b_i^{EE}$	total of EE request blocking bounds
$b_i^{CE}$	total of CE request blocking bounds
$b_i^E$	total of engine lock request blocking bounds
$N_i^I$	number of chunks to transmit $z_i^I$
$N_i^O$	number of chunks to transmit $z_i^O$
$N_i^S$	number of chunks to transmit $z_i^S$
$X^I$	time to transmit a chunk to a GPU from system memory
$X^O$	time to transmit a chunk to system memory from a GPU
$X^{P2P}$	time to transmit a chunk through peer-to-peer DMA
$X^{max}$	maximum of $X^I$ , $X^O$ , and $X^{P2P}$
$S_i$	time needed to perform a GPU migration
$\mathcal{R}_i^k$	set of requests, sorted by $L_j^k$ , for resource $\ell_k$ that may interfere with a request of job $J_i$ for $\ell_k$
$\mathcal{R}_i^{EE}$	set of EE requests, sorted by $L_j^k$ , that may interfere with a similar request of job $J_i$
$\mathcal{R}_i^{CE}$	set of CE requests, sorted by $L_j^k$ , that may interfere with a CE request of job $J_i$
Overhead-Aware Analysis Parameters	
$\lambda$	number of GPU engines of a GPU
$\Delta^{top}$	GPU interrupt top-half overhead
$\Delta^{bot}$	GPU interrupt bottom-half overhead
$H_i$	number of GPU interrupts that may interfere with job $J_i$
$\gamma_i$	number of GPU engine operations issued by job $J_i$

Table 4.1: Summary of additional parameters to describe and analyze sporadic task sets with GPUs.

then derive *coarse-grain* bounds on pi-blocking due to engine lock requests.<sup>7</sup> We call this analysis “coarse” because it assumes that all engine critical sections have the same length. Although we do not use coarse analysis in our own schedulability experiments (presented later in Section 4.3.6), this level of analysis gives us an appreciation for the order of complexity of engine lock pi-blocking, in terms of the *number* of interfering requests, and the tradeoffs between GPU migration through system memory and direct peer-to-peer DMA. We then delve into detailed, or “fine-grain,” analysis to bound pi-blocking caused by engine lock and token requests.

#### 4.3.4.1 Three-Phase Blocking Analysis

The maximum total time a job may be pi-blocked accessing tokens and engine locks is given by the equation

$$b_i = b_i^E + b_i^K. \quad (4.4)$$

Our challenge is to determine pi-blocking bounds on  $b_i^E$  and  $b_i^K$ .

We approach this problem using a three-phase process. In the first phase, we compute  $b_i^E$  for each task  $T_i \in \mathcal{T}$ . In the second phase, we use the computed bounds on engine lock pi-blocking to bound the length of the token critical section of each GPU-using task, denoted by  $L_i^K$ . More precisely, we bound  $L_i^K$  with the following equation:

$$L_i^K = q_i + xmit(z_i^I, z_i^O, z_i^S) + e_i^{gpu} + b_i^E. \quad (4.5)$$

That is,  $L_i^K$  is bounded by the sum of: **(i)** the total time job  $J_i$  executes on an EE ( $q_i$ ); **(ii)** the time to perform all possible DMA operations ( $xmit(z_i^I, z_i^O, z_i^S)$ ); **(iii)** the CPU execution time of  $J_i$  that occurs within the token critical section ( $e_i^{gpu}$ ); and finally, **(iv)** the total time  $J_i$  may be pi-blocked waiting for engine locks. In the third phase, we bound pi-blocking induced by GPU token requests using analysis appropriate for the GPU Allocator configuration.

#### 4.3.4.2 Coarse-Grain Blocking Analysis for Engine Locks

We now derive coarse-grain pi-blocking bounds for FIFO-ordered engine locks.

---

<sup>7</sup>We do not present coarse-grain bounds for token requests in this chapter, since we have already presented this analysis in Sections 2.1.7.2 and 2.1.7.3 for the relevant GPU Allocator configurations.

Let  $b_i^{EE}$  denote  $T_i$ 's maximum total pi-blocking time for the EE lock, let  $b_i^{I/O}$  denote its maximum total pi-blocking time while waiting to transmit input and output chunks, and let  $b_i^{P2P}$  denote its maximum total pi-blocking time while waiting for CE locks to perform a peer-to-peer migration. By construction,

$$b_i = b_i^{EE} + b_i^{I/O} + b_i^{P2P}. \quad (4.6)$$

A job may be pi-blocked for every GPU kernel it executes when acquiring the EE lock of its allocated GPU. At most  $\rho - 1$  other jobs may compete simultaneously for this lock for a given request. Since requests are FIFO-ordered, the resulting pi-blocking is bounded by

$$b_i^{EE} = (\rho - 1) \cdot \max\{q_j \mid T_j \in \mathcal{T}_a^{gpu}\}, \quad (4.7)$$

where task  $T_i$  is assigned to the  $a^{th}$  GPU cluster.

Bounds for  $b_i^{I/O}$  and  $b_i^{P2P}$  depend partly on whether migrations are peer-to-peer or through system memory. In our analysis, we assume that all migrations are performed using the same method, though GPUSync can support both types in the same system.

**CE blocking with peer-to-peer.** Under peer-to-peer migrations, any task holding a GPU token may request the CE lock of the GPU it used in its prior job in order to perform a migration. There are at most  $\rho \cdot g$  such tasks. In the worst case, they may all attempt to access the same CE lock at the same instant. Thus, *any* request for a CE lock may be blocked by  $(\rho \cdot g - 1)$  other requests. From the blocking analysis of DGLs of Ward and Anderson (2013), the total number of interfering requests for a CE is at most  $(\rho \cdot g - 1)$ . Since no request requires more than  $X^{max}$  time to complete,

$$b_i^{I/O} = X^{max} (N_i^I + N_i^O) (\rho \cdot g - 1) \quad (4.8)$$

and

$$b_i^{P2P} = X^{max} \cdot N_i^S (\rho \cdot g - 1). \quad (4.9)$$

**CE blocking with system memory migration.** When migration between GPUs takes place via system memory, CEs are only accessed by tasks that have been given a token for an allocated GPU, so at most  $\rho - 1$  other jobs may compete for the CE lock at a given instant. Recall from Section 3.2.3.4 that state is aggregated

with input and output data, in this case. Thus,  $b_i^{P2P} = 0$ . However, now

$$b_i^{I/O} = X^{max}(N_i^I + N_i^O + 2 \cdot N_i^S)(\rho - 1), \quad (4.10)$$

since state data must be handled twice.

Analytical bounds for peer-to-peer and system memory migrations differ. As seen above, CE lock contention is  $\mathcal{O}(\rho \cdot g)$  and  $\mathcal{O}(\rho)$  under peer-to-peer and system memory migrations, respectively. Despite its inferior order of complexity, peer-to-peer migration may still result in better analytical bounds *if* the advantages of fewer total DMA operations and faster peer-to-peer DMA operations can be exploited. Also, there are benefits to peer-to-peer migrations that are not captured in the above analysis, namely, isolation from the system memory bus and rarity of migrations due to the GPU Allocator’s heuristics.

#### 4.3.4.3 Detailed Blocking Analysis for Engine Locks

Our detailed blocking analysis, whether for engine lock or token requests, follows the same general approach, which we outline before delving into detailed analysis.<sup>8</sup> For task  $T_i$ , we first determine the *number* of jobs of another task,  $T_j$  where  $i \neq j$ , that may be ready to run at the same time as  $J_{i,u}$ . This is characterized by the *task interference function*,  $tif(T_i, T_j)$ . From  $tif(T_i, T_j)$ , we generate a set of interfering resource requests that the interfering jobs  $J_{j,v}$  may make when  $J_{i,u}$  requests a resource of the same *type*, where type may be GPU token, execution engine lock, or copy engine lock. This set is generated by the *request interference function*,  $xif(T_i, T_j, \ell_k)$ , where  $\ell_k$  is a given resource. We aggregate the set of interfering requests of all tasks (excluding  $T_i$ ) into a single set of *all* interfering resource requests that may be made, as given by the *total request interference function*,  $txif(T_i, \ell_k)$ .

Each interfering request,  $R_j$ , has an associated length,  $L_j$ . The set defined by  $\mathcal{R}_i^k \triangleq txif(T_i, \ell_k)$  is *sorted* in descending order by length. To compute the pi-blocking experienced by  $J_{i,u}$  for a given single resource request, the top  $y$  requests are removed from  $\mathcal{R}_i^k$ , depleting  $\mathcal{R}_i^k$  by  $y$  requests, and summed. This process is repeated iteratively for each request of a given type made by  $J_{i,u}$ , terminating early if  $\mathcal{R}_i^k$  becomes empty. In general, the value of  $y$  depends upon the locking protocol used and resource organization. For example, under GPUSync,  $y = (\rho \cdot g - 1)$  for CE lock requests when peer-to-peer migrations are used. (This is derived directly

---

<sup>8</sup>We follow the general approach used by Brandenburg (2011b), so we adopt his terminology and formulas for modeling the set of requests that may interfere with a request issued by a job of task  $T_i$ .

from the blocking complexity of peer-to-peer migrations, which we discussed at the end of Section 4.3.4.2.) This entire process must be repeated for each resource: GPU token, EE lock, and CE lock(s).

In the case of soft real-time scheduling,  $tif(T_i, T_j)$  depends upon tardiness bounds, which in turn depend upon blocking bounds. A fixed-point iterative method must then also be used ensure schedulability, outlined by the following steps:

1. Initialize tardiness bounds to zero.
2. Compute pi-blocking bounds.
3. Compute tardiness bounds.
4. Compute new pi-blocking bounds, incorporating tardiness bounds.
5. Check schedulability. Go back to step (3) if the task set is schedulable, but new pi-blocking bounds from step (4) differ from bounds previously computed.

This method will terminate when either bounds on pi-blocking have reached a steady state or the task set is unschedulable. This highlights a significant benefit of FL scheduling over EDF scheduling: *the tighter tardiness bounds offered by FL scheduling may reduce computed pi-blocking bounds.*

Before proceeding, we state two important assumptions. First, we assume that the total number of GPU-using tasks is greater than  $g$ ; otherwise, GPU Allocator Heuristics H1 and H2 load-balance GPU token requests such that no two tasks share a GPU *and* that each task always receives the same GPU for every job—there is no blocking or migration under this scenario. Second, we assume that a GPU-using task only requests a GPU token once per job. The following analysis can be extended to handle multiple token requests per job, though it becomes cumbersome to express. We now proceed to define the above formulas. We direct the reader towards the work of Brandenburg (2011b) for a detailed explanation of each formula.

**Definition 4.1.** For hard real-time systems,

$$tif(T_i, T_j) \triangleq \left\lceil \frac{p_i + r_j}{p_j} \right\rceil, \quad (4.11)$$

**Definition 4.2.** For soft real-time systems (under the “bounded tardiness” definition of soft real-time),

$$tif(T_i, T_j) \triangleq \left\lceil \frac{\max(p_i, r_i) + r_j}{p_j} \right\rceil. \quad (4.12)$$

$tif(T_i, T_j)$  gives us the number of jobs of  $T_j$  that may interfere with a job  $J_{i,u}$ . We use a different definition of  $tif(T_i, T_j)$  (Equation (4.12)) for soft real-time systems, since a job of  $T_i$  may be tardy—we must consider a larger window of execution in which such a job may issue resource requests. We now derive the set of requests from  $T_j$  that may interfere with requests of  $J_{i,u}$  for resource  $\ell_k$ .

**Definition 4.3.** The set of requests of  $T_j$  that interfere with requests of a job of  $T_i$  for resource  $\ell_k$  is given by

$$xif(T_i, T_j, \ell_k) \triangleq \{R_{j,v} \mid 1 \leq v \leq tif(T_i, T_j) \cdot \eta_{j,k}\}, \quad (4.13)$$

where  $\eta_{j,k}$  is the maximum number of requests for  $\ell_k$  that a job of  $T_j$  may make.

We say that  $xif(T_i, T_j, \ell_k)$  defines a set of *generic* requests because request  $R_{j,v} \in xif(T_i, T_j, \ell_k)$  does not denote the  $v^{th}$  request made by task  $T_j$  after the release of  $T_j$ 's first job. Rather,  $R_{j,v}$  denotes the  $v^{th}$  resource request in a worst-case string of consecutive requests of  $T_j$  that may interfere with request  $R_i$  of  $T_i$ .

Finally, we can derive an aggregate of all interfering requests.

**Definition 4.4.** The set of all interfering resource requests of other jobs that may interfere with requests of a job of  $T_i$  for resource  $\ell_k$  is given by

$$txif(T_i, \ell_k) \triangleq \bigcup_{T_j \in \mathcal{T} \setminus \{T_i\}} xif(T_i, T_j, \ell_k). \quad (4.14)$$

We these formulas defined, we can now present detailed pi-blocking analysis for engine lock and token requests.

**Detailed analysis for execution engine lock requests.** We can now compute a bound on worst-case pi-blocking job  $J_i$  experiences when it requests an EE,  $b_i^{EE}$ . Let the resource  $\ell_k$  represent a particular execution engine lock. Let the function  $top(v, \mathcal{R}_i^k)$  denote the  $v$  longest requests in the set of requests  $\mathcal{R}_i^k$  for  $\ell_k$ . The set  $\mathcal{R}_i^k$  is given by  $txif(T_i, \ell_k)$ , by construction. The total worst-case pi-blocking experienced by job  $J_i$  while waiting for an execution engine is bounded by

$$b_i^{EE} = \sum_{R_j \in top((\rho-1) \cdot \eta_{i,k}, \mathcal{R}_i^k)} L_j, \quad (4.15)$$

where  $\eta_{i,k}$  denotes the number of EE requests issued by job  $J_i$  for  $\ell_k$ .

**Detailed analysis for copy engine lock requests with peer-to-peer migration.** In Section 4.3.4.2, we computed pi-blocking for kernel input/output DMA operations and peer-to-peer DMA operations separately, denoted by the terms  $b_i^{I/O}$  and  $b_i^{P2P}$ , respectively. Under detailed analysis, it is easier to compute bounds on pi-blocking for these different types of DMA operations *jointly*. We denote pi-blocking due to all CE requests by  $b_i^{CE}$ . We first present detailed analysis that holds for GPUs with either one or two CEs. We then present tighter analysis for the dual-CE case.

Let  $\ell^I$ ,  $\ell^O$ , and  $\ell^{P2P}$ , represent the CE(s) of the same single GPU used by job  $J_i$  to transmit kernel input to a GPU, transmit kernel output from a GPU, and migrate state to a GPU, respectively. The CE(s) represented by these resources may be one in the same. However, we generate the set of interfering requests by considering them as separate resources, each dedicated to performing a particular type of DMA (*e.g.*, input, output, or peer-to-peer migration). Let  $\mathcal{R}_i^I$  denote the sorted set of requests for copying kernel input data to a GPU that may interfere with a similar request of job  $J_i$ ;  $\mathcal{R}_i^I \triangleq \text{txif}(T_i, \ell^I)$ . Let  $\mathcal{R}_i^O$  denote the sorted set of requests for copying kernel output data from a GPU that may interfere with a similar request of job  $J_i$ ;  $\mathcal{R}_i^O \triangleq \text{txif}(T_i, \ell^O)$ . Let  $\mathcal{R}_i^{P2P}$  denote the sorted set of requests of peer-to-peer DMA requests that may interfere with a similar request of job  $J_i$ ;  $\mathcal{R}_i^{P2P} \triangleq \text{txif}(T_i, \ell^{P2P})$ .  $\mathcal{R}_i^{P2P}$  includes interfering requests of jobs that may migrate to *and from* the GPU allocated to job  $J_i$ . The sorted set of all CE requests that may interfere with a CE request of job  $J_i$  is denoted by  $\mathcal{R}_i^{CE}$ :  $\mathcal{R}_i^{CE} \triangleq \mathcal{R}_i^I \cup \mathcal{R}_i^O \cup \mathcal{R}_i^{P2P}$ .

The total worst-case pi-blocking experienced by  $J_i$  while waiting to receive the requested CE lock when peer-to-peer migrations are used is bounded by

$$b_i^{CE} = \sum_{R_j \in \text{top}((\rho \cdot g - 1) \cdot (N_i^I + N_i^O + N_i^S), \mathcal{R}_i^{CE})} L_j, \quad (4.16)$$

where  $L_j$  is equal to the length of the associated CE operation (*e.g.*,  $X^I$ ,  $X^O$ , or  $X^{P2P}$ ).

Observe that the above analysis does not take advantage of the fact that the GPU has two CEs. That is, the above analysis holds when a GPU has only *one* CE. The transitive pi-blocking induced by peer-to-peer migrations makes it difficult to derive tighter bounds for dual-CE GPUs. However, tighter analysis is possible. We now describe this optimization.

Transitive pi-blocking due to peer-to-peer migrations is only possible when  $\mathcal{R}_i^{P2P} \neq \emptyset$ . Recall that the computation of  $b_i^{CE}$  is iterative: requests from  $\mathcal{R}_i^{CE}$  are *extracted* in groups of  $\rho \cdot g - 1$  at a time.

Let  $\widehat{\mathcal{R}}_i^{CE,k}$  denote the set of requests remaining after the  $k^{th}$  iteration of  $b_i^{CE}$ 's computation. Let  $\widehat{\mathcal{R}}_i^{I,k} \triangleq \widehat{\mathcal{R}}_i^{CE,k} \cap \mathcal{R}_i^I$ , denoting the sorted set of *remaining* interfering input requests in  $\widehat{\mathcal{R}}_i^{CE,k}$ . Let  $\widehat{\mathcal{R}}_i^{O,k} \triangleq \widehat{\mathcal{R}}_i^{CE,k} \cap \mathcal{R}_i^O$ , denoting the sorted set of *remaining* interfering output requests in  $\widehat{\mathcal{R}}_i^{CE,k}$ . Let  $\widehat{\mathcal{R}}_i^{P2P,k} \triangleq \widehat{\mathcal{R}}_i^{CE,k} \cap \mathcal{R}_i^{P2P}$ , denoting the sorted set of *remaining* interfering migration requests in  $\widehat{\mathcal{R}}_i^{CE,k}$ . If  $\widehat{\mathcal{R}}_i^{P2P,k} = \emptyset$  then transitive pi-blocking due to migrations is no longer possible, since no migration requests remain. From this observation, we derive the following tighter analysis for dual-CE GPUs where  $b_i^{CE}$  is broken down into two terms,  $b_i^{CE_{trans}}$  and  $b_i^{CE_{direct}}$ , where pi-blocking complexity is  $\mathcal{O}(\rho \cdot g)$  and  $\mathcal{O}(\rho)$ , respectively. Let  $\kappa \in \mathbb{N}_1$  denote the smallest integer such that  $(\mathcal{R}_i^{CE} \setminus \text{top}((\rho \cdot g - 1) \cdot \kappa, \mathcal{R}_i^{CE})) \cap \mathcal{R}_i^{P2P} = \emptyset$ . Observe that  $(\mathcal{R}_i^{CE} \setminus \text{top}((\rho \cdot g - 1) \cdot \kappa, \mathcal{R}_i^{CE})) \equiv \widehat{\mathcal{R}}_i^{CE,\kappa}$ .

The total *transitive* worst-case pi-blocking experienced by  $J_i$  while waiting for a CE to copy data to or from a GPU when peer-to-peer migrations are used with dual-CE GPUs is bounded by

$$b_i^{CE_{trans}} = \sum_{R_j \in \text{top}((\rho \cdot g - 1) \cdot \kappa, \mathcal{R}_i^{CE})} L_j. \quad (4.17)$$

The total *direct* worst-case pi-blocking experienced by  $J_i$  while waiting for a CE to copy data to or from a GPU when peer-to-peer migrations are used with dual-CE GPUs is bounded by

$$b_i^{CE_{direct}} = \sum_{R_j \in \text{top}((\rho - 1) \cdot \max(N_i^I + N_i^O + N_i^S - \kappa, 0), \widehat{\mathcal{R}}_i^{CE,\kappa})} L_j. \quad (4.18)$$

By construction, the total worst-case pi-blocking experienced by  $J_i$  while waiting for a CE to copy data to or from a GPU when peer-to-peer migrations are used with dual-CE GPUs is bounded by

$$b_i^{CE} = b_i^{CE_{trans}} + b_i^{CE_{direct}}. \quad (4.19)$$

**Blocking chains.** Are tighter bounds on CE pi-blocking possible? Certainly. We call a sequence of CE requests that may block a request of job  $J_i$  a *blocking chain*. The number of all possible blocking chains is finite, since each token holder may issue at most one DMA operation at a time. When  $g$  and  $\rho$  are relatively small, it is feasible to enumerate each potential blocking chain. For example, Table 4.2 depicts all possible *representative* blocking chains for *outbound*, *inbound*, and *migration* CE requests when  $g = 2$ ,  $\rho = 3$ , and all GPUs have two CEs. In the table, “O,” “I,” and “M” represent outbound, inbound, and migration DMA



Outbound		Inbound		Migration	
Chain	Cost ( $\mu$ s)	Chain	Cost ( $\mu$ s)	Chain	Cost ( $\mu$ s)
IIOOM	4967	IOOOM	5000	IIOOO	6006
IOOMM	3961	OOOMM	3994	IOOOM	5000
IOOM	3786	OOOM	3818	IIOOO	4824
IIOIM	3753	IOOM	3786	IIOO	4791
OOMMM	2956	OOMM	2780	OOOMM	3994
OOMM	2780	OOM	2604	OOOM	3818
IOMM	2747	IOM	2571	IOOM	3786
OOM	2604	II	2363	OOO	3643
IOM	2571	OMM	1566	IOO	3610
IIM	2538	OM	1390	IIO	3577
OO	2429	IM	1357	OOMM	2780
OMMM	1741	I	1181	OOM	2604
OMM	1566	MM	351	IOM	2571
IMM	1533	M	176	OO	2429
OM	1390	—	—	IO	2396
IM	1357	—	—	II	2363
O	1214	—	—	OMM	1566
MMM	527	—	—	OM	1390
MM	351	—	—	IM	1357
M	176	—	—	O	1214
—	—	—	—	I	1181
—	—	—	—	MM	351
—	—	—	—	M	176

Table 4.2: All possible representative blocking chains that may delay a CE request for an outbound, inbound, or migration operation for GPUs with two CEs,  $g = 2$ , and  $\rho = 3$ . Costs computed assuming worst-case conditions for a 1MB DMA operation and non-interleaved system memory.

operations, respectively. An outbound DMA operation is one where a CE is used to copy data away from a GPU. Likewise, an inbound DMA operation is one where a CE is used to copy data onto a GPU. A migration DMA operation is simultaneously an inbound and an outbound DMA operation. Each letter (“I,” “O,” or “M”) represents a type of CE request that interferes with a CE request of job  $J_i$ . We say the chains in Table 4.2 are “representative,” since there may exist multiple DMA sequences (or “actual” blocking chains) that contain the same frequency of request types, but differ in the order these requests appear within each sequence and which particular GPUs handle each request. We derive the cost of each blocking chain (*i.e.*, the time job  $J_i$  can be blocked by said chain) by summing the costs of the individual DMA operations within each chain. Thus, blocking chains that have the same frequency of request types also have the same cost. We can identify such requests with a single representative blocking chain. For example, the chains “IMO” and “MOI” each contain one outbound, inbound, and migration operation, so these chains have the same cost. In Table 4.2, we take the chain “IMO” to represent all equivalent chains.

The chains in Table 4.2 are sorted in order of decreasing cost. Here, we assume that all DMA operations are 1MB in size. The costs of individual DMA operations are derived from our overhead model where we assume worst-case overheads in a loaded system with non-interleaved system memory. We also assume that all peer-to-peer migrations are “near,” as we discussed in Section 4.3.1.2, since  $g = 2$ .

Table 4.2 is divided into outbound, inbound, and migration columns to denote the sets of possible blocking chains that may interfere with a CE request of job  $J_i$  for outbound, inbound, and migration DMA operations, respectively. Observe that no chain in Table 4.2 has a length of more than five operations. This is consistent with the CE blocking bounds we derived in Section 4.3.4.2 for system configurations with peer-to-peer migration, where we showed that each CE lock request is blocked by at most  $\rho \cdot g - 1$  CE requests:  $2 \cdot 3 - 1 = 5$ .

Although some representative chains appear in all three columns (*e.g.*, “IOOM”), others do not. For example, the chain “OO” does not appear in the inbound column. This is because it is impossible for an inbound CE lock request to be blocked exclusively by outbound CE lock requests, since GPUSync directs inbound and outbound DMA operations to different CEs when a GPU has two CEs. A blocking chain where an inbound CE lock request is blocked by an outbound CE lock request *must* include a migration CE lock request in order to link the otherwise independent CEs.

Under GPUSync, peer-to-peer migrations are *pulled* from one GPU to another. That is, the job that issues a migration CE request always holds a token for the *destination* GPU of the peer-to-peer DMA operation.

A peer-to-peer migration always uses the inbound CE of the requesting job’s assigned GPU. It is for this reason that the set of blocking chains for inbound requests is a proper subset of the set of blocking chains for migration requests. However, we cannot treat migration requests as inbound requests. This is because a migration request may also contend with requests for the outbound CE of the source GPU of the peer-to-peer DMA operation (even if the request experiences no contention for the inbound CE of the destination GPU).

The pull-based nature of peer-to-peer migrations also affects the blocking chains associated with outbound CE requests. There may be at most  $\rho$  simultaneous inbound migration requests for GPU<sub>*a*</sub>’s inbound CE, since only jobs assigned a token for GPU<sub>*a*</sub> ever access GPU<sub>*a*</sub>’s inbound CE. However, there may be as many as  $\rho \cdot (g - 1)$  simultaneous *outbound* migration requests for GPU<sub>*a*</sub>’s outbound CE. It is for this reason that the inbound and outbound columns of Table 4.2 differ.

**Computing all possible blocking chains.** In support of the schedulability experiments we present in Section 4.3.6, we computed tables of all possible blocking chains for system configuration defined by the unique combinations of system configuration parameters  $g \in \{2, 4\}$  and  $\rho \in \{1, 2, 3\}$ , for systems with dual-CE GPUs.<sup>9</sup> We use a brute-force algorithm to perform an exhaustive search of all possible blocking chains for outbound, inbound, and migration requests.

To compute the blocking chains for a given CE request of job  $J_i$  on GPU<sub>*a*</sub>, we consider the situation where all token holders may have incomplete CE requests that were issued prior to that of  $J_i$ ’s. Job  $J_j$ , assigned to GPU<sub>*b*</sub>, may have issued an outbound, inbound, or migration request from any of the other  $g - 1$  GPUs. Job  $J_j$  may also have not issued a request, which we represent with a place-holder “null” request. Thus,  $J_j$  may perform one of  $(1 + 1 + (g - 1) + 1) = g + 2$  possible operations. Since there are  $\rho \cdot g - 1$  token holders, excluding job  $J_i$  in the cluster that includes GPU<sub>*a*</sub>, there may be as many as  $(\rho \cdot g - 1)(g + 2)$  different sets of incomplete requests issued before job  $J_i$ ’s request. The order in which these requests are issued may affect the blocking experienced by job  $J_i$ . There are  $(\rho \cdot g - 1)!$  different ways in which we may order the requests (including null requests) in each of these sets. This results in  $(\rho \cdot g - 1)(g + 2)((\rho \cdot g - 1)!)$  scenarios in which these requests issued by the token holders other than job  $J_i$  may precede the request of job  $J_i$ .

---

<sup>9</sup>We do not compute tables for configurations where  $g \in \{1, 8\}$ . Peer-to-peer migrations are not used when  $g = 1$  (*i.e.*, a partitioned GPU configuration). On our evaluation platform, peer-to-peer migrations are not possible when  $g = 8$ , since peer-to-peer DMA operations cannot cross NUMA boundaries.

Request	$R_1$	$R_2$	$R_3$	$R_4$	$R_5$
Request Type	I	M	O	O	N
GPU(s) Handling Request	GPU <sub>0</sub>	GPU <sub>0</sub> $\leftarrow$ GPU <sub>1</sub>	GPU <sub>1</sub>	GPU <sub>1</sub>	—
GPU Assigned to Requesting Task	GPU <sub>0</sub>	GPU <sub>0</sub>	GPU <sub>1</sub>	GPU <sub>1</sub>	GPU <sub>1</sub>

Table 4.3: A possible arrangement of copy engine requests where  $g = 2$  and  $\rho = 3$ .

With a procedure we discuss shortly, we evaluate each of these scenarios to construct a blocking chain for each type of request that may be issued by job  $J_i$ , and from it, we construct a representative blocking chain. We insert this representative blocking chain into a hash table if it is not already stored therein. This algorithm leaves *much* to be desired. It inspects  $(\rho \cdot g - 1)(g + 2)((\rho \cdot g - 1)!)$  scenarios. The procedure we describe below takes  $\mathcal{O}(\rho \cdot g)$  to evaluate each scenario. As a result, our brute-force algorithm has a runtime complexity of  $\mathcal{O}((\rho^2 \cdot g^3)((\rho \cdot g - 1)!))$ . Still, we find that this algorithm takes no more than several hours for small values of  $g$  and  $\rho$ , where  $g \leq 3$  and  $\rho \leq 3$ .<sup>10,11</sup> This is acceptable, since the results can be stored offline and reused in schedulability experiments.

We now describe the process we use to compute a blocking chain for each request scenario. Each scenario is represented by a string of requests, where each type of request is denoted by a symbol (*i.e.*, O, I, M, or N (for null requests)). Each request is paired with a GPU identifier, indicating which GPU is to handle the request. We interpret the order in which these requests are arranged, left to right, as a temporal ordering—the most recent request is at the head of the string, and the least recent request at the tail. The row labeled “Request Type” in Table 4.3 depicts such a string for a platform where  $g = 2$  and  $\rho = 3$ : {I, M, O, O, N}. Table 4.3 also includes information about each request, such as the GPU handling each request and the GPU to which the requesting task is assigned. Recall that migration requests are actually *two* copy engine requests that are issued simultaneously and atomically through a DGL request (please refer to Sections 2.1.6.2 and 3.3.3 for details). This allows us to treat these dual-requests as a single request that spans two GPUs. Request  $R_2$  in Table 4.3 is such request. We use an arrow to describe the migration between GPUs. For example, request  $R_2$  is for a migration from GPU<sub>1</sub> to GPU<sub>0</sub>, which is denoted by “GPU<sub>0</sub>  $\leftarrow$  GPU<sub>1</sub>” in the row labeled “GPU(s) Handling Request” in Table 4.3.

<sup>10</sup>When  $g = 4$  and  $\rho = 3$ , we must make optimizations that allow us to consolidate computations for scenarios with common requests in order to complete within a reasonable time frame.

<sup>11</sup>The code for the implementation of our algorithm is freely available at [www.github.com/GElliott](http://www.github.com/GElliott).

Every prefix-substring of the string for a given scenario *may* be a chain of requests that blocks a request issued by job  $J_i$ . Whether a chain of requests actually blocks a request of job  $J_i$  partly depends upon the request type issued by  $J_i$  and to which GPU(s) the request is directed. Without loss of generality, we assume every request issued by  $J_i$ , denoted by  $R_0$ , is for an inbound or outbound copy engine of GPU<sub>0</sub>. In the case of migrations, we assume job  $J_i$  migrates away from GPU<sub>1</sub> to GPU<sub>0</sub>. This is a safe generalization because our ultimate goal is to find all *representative* blocking chains for a particular platform configuration. Since GPUs are homogeneous under GPUSync, the set of actual blocking chains that may block a request of a given type for GPU<sub>0</sub> is homomorphic to the set of actual blocking chains that block a request of the same type for GPU<sub>1</sub>. Representative blocking chains only describe the frequency of each request type in a chain; information on the order in which requests are issued, as well as to which GPU each request is issued, is stripped away when a representative chain is constructed from an actual chain.

We use the recursive procedure COMPUTEBLOCKINGCHAIN, depicted in Figure 4.12(a), to compute a blocking chain for a given scenario and initial request type. The procedure inspects the string of requests, denoted by the function parameter *requestString*, and returns the maximal prefix-substring, in terms of string length, that blocks the request on the top of the request stack, denoted by the function parameter *stack*. We initialize the parameter *stack* with the request of job  $J_i$  prior to the first call to COMPUTEBLOCKINGCHAIN, as seen in insets (b), (c), and (d) of Figure 4.12, for inbound, outbound, and migration requests, respectively. We now describe COMPUTEBLOCKINGCHAIN in more detail. If *requestString* is an empty string, then no request in *stack* can be blocked. In this case (when line 2 evaluates to *false*), COMPUTEBLOCKINGCHAIN returns a empty blocking chain at line 18. Otherwise, the procedure pops a request from *stack* (line 3) and stores it in the variable *request*. The parameter *stack* is guaranteed to contain at least one request at line 3, since a request is pushed onto the stack immediately prior to a call to COMPUTEBLOCKINGCHAIN (e.g., line 6), or *stack* is known to contain at least one request prior to calling COMPUTEBLOCKINGCHAIN (e.g., line 12). After extracting a request from *stack*, the procedure obtains a reference to the first element in *requestString* (line 4); this request is stored in the variable *next*. We use the subroutine BLOCKEDBY to test whether *request* is blocked by *next* (line 5). BLOCKEDBY returns *true* if requests *request* and *next* contend for the same CEs; it returns *false*, otherwise. If *next* blocks *request*, then we push *next* onto *stack* (line 6), and we recursively call COMPUTEBLOCKINGCHAIN (line 11) to process the requests that follow *next* in *requestString* (the substring of *requestString* that makes up these requests is denoted by *requestString*[1:]). At line 11, the procedure joins two lists of requests to create the returned blocking chain. The first list is

```

1: procedure COMPUTEBLOCKINGCHAIN(stack, requestString)
2:   if requestString  $\neq \emptyset$  then
3:     request  $\leftarrow$  POP(stack) ▷ Pop request from stack.
4:     next  $\leftarrow$  requestString[0] ▷ Process request at head of requestString.
5:     if BLOCKEDBY(request, next) then ▷ True if next and request contend for same CE locks.
6:       PUSH(stack, next) ▷ Prepare to compute chain of what may block next.
7:       if ISMIGRATIONREQUEST(next) then
8:         PUSH(stack, next) ▷ Push next twice, since migrations are two requests.
9:       end if
10:      ▷ Recurse on next and remainder of requestString.
11:      return {next} + COMPUTEBLOCKINGCHAIN(stack, requestString[1 :])
12:    else if stack not empty then
13:      ▷ Migration request at top of stack may be blocked by next; recurse.
14:      return COMPUTEBLOCKINGCHAIN(stack, requestString)
15:    end if
16:  end if
17:  ▷ Return nothing if (a) no more requests or (b) next does not block request and stack is empty.
18:  return {}
19: end procedure

```

(a) Recursive procedure for computing a blocking chain for a given scenario.

```

1: procedure COMPUTEINBOUNDLOCKINGCHAIN(scenario)
2:   stack  $\leftarrow$  { CREATEINBOUNDREQUEST(GPU0) } ▷ Initialize a stack with request on top.
3:   return COMPUTEBLOCKINGCHAIN(stack, scenario)
4: end procedure

```

(b) Computes the blocking chain for an inbound request.

```

1: procedure COMPUTEOUTBOUNDLOCKINGCHAIN(scenario)
2:   stack  $\leftarrow$  { CREATEOUTBOUNDREQUEST(GPU0) } ▷ Initialize a stack with request on top.
3:   return COMPUTEBLOCKINGCHAIN(stack, scenario)
4: end procedure

```

(c) Computes the blocking chain for an outbound request.

```

1: procedure COMPUTEMIGRATIONBLOCKINGCHAIN(scenario)
2:   request  $\leftarrow$  CREATEMIGRATIONREQUEST(GPU0, GPU1) ▷ Migration from GPU1 to GPU0.
3:   ▷ Initialize stack with request twice, since migrations are two requests.
4:   stack  $\leftarrow$  {request, request}
5:   return COMPUTEBLOCKINGCHAIN(stack, scenario)
6: end procedure

```

(d) Computes the blocking chain for a migration request.

Figure 4.12: Procedures for computing blocking chains for a given request scenario.

composed of a single element, constructed from *next*. The second list is whatever is returned by the recursive call to COMPUTEBLOCKINGCHAIN. The remaining lines of code in COMPUTEBLOCKINGCHAIN handle a special case introduced by migration requests, which we discuss next.

Consider the following hypothetical situation. Suppose we *removed* lines 7, 8, and 12 through 14 from COMPUTEBLOCKINGCHAIN, and the procedure were called with  $stack = \{(M, GPU_0 \rightarrow GPU_1)\}$  and  $requestString = \{(I, GPU_0), (O, GPU_1)\}$ . The broken procedure would correctly find that the request  $(I, GPU_0)$  blocks  $(M, GPU_0 \rightarrow GPU_1)$ , since both requests contend for the inbound CE of  $GPU_0$ . In the subsequent recursive call made from line 11,  $stack = \{(I, GPU_0)\}$  and  $requestString = \{(O, GPU_1)\}$ . As result, our broken procedure would determine that request  $(I, GPU_0)$  is not blocked by  $(O, GPU_1)$  (which is true), and the call would return the empty chain on line 18. Our broken version of COMPUTEBLOCKINGCHAIN would return the chain  $\{(I, GPU_0)\}$  for request  $(M, GPU_0 \rightarrow GPU_1)$ . This is incorrect. The request  $(M, GPU_0 \rightarrow GPU_1)$  may be blocked by *both* requests  $(I, GPU_0)$  and  $(O, GPU_1)$ , since the migration request requires the inbound CE of  $GPU_0$  and the outbound CE of  $GPU_1$  in order to proceed. Recall from Section 2.1.6.2 that only one job of the two jobs associated with requests  $(I, GPU_0)$  and  $(O, GPU_1)$  may inherit the priority of the job that issued request  $(M, GPU_0 \rightarrow GPU_1)$  at a time. Even though the DMA operations of requests  $(I, GPU_0)$  and  $(O, GPU_1)$  potentially execute in parallel, in the worst-case, requests  $(I, GPU_0)$  and  $(O, GPU_1)$  complete serially. Hence, request  $(M, GPU_0 \rightarrow GPU_1)$  may be blocked by both requests  $(I, GPU_0)$  and  $(O, GPU_1)$ , one after the other. In other words, request  $(M, GPU_0 \rightarrow GPU_1)$  may be blocked by two parallel blocking chains, one starting from  $GPU_0$  and the other starting from  $GPU_1$ , that are encountered serially.

To correctly handle this case, COMPUTEBLOCKINGCHAIN checks whether a migration request is blocked on both its source and destination GPUs. This is accomplished by pushing a migration request onto *stack* a second time (e.g., at line 8 in Figure 4.12(a), as well as on line 4 in Figure 4.12(d)). This realizes the following behavior: if COMPUTEBLOCKINGCHAIN reaches the potential end to a blocking chain, the procedure “rolls back” to a prior encountered migration request that may still be blocked by requests in *requestString*, and attempts to construct a parallel blocking chain. This parallel chain is naturally appended to the already constructed partial chain. This roll-back-and-continue behavior is implemented through lines 12 through 14. At line 12, the procedure knows that *request* is not blocked by *next*. If *stack* is not empty, then the procedure checks whether *next* blocks the newly exposed request on the top of *stack*—this request must

be a migration request, because all non-migration requests, which may only appear in *stack* once, are popped from *stack* immediately after they are pushed onto *stack* (if requests in *requestString* remain to be evaluated).

Let us reconsider the prior hypothetical situation under the correct version of COMPUTEBLOCKINGCHAIN. COMPUTEBLOCKINGCHAIN is called with  $stack = \{(M, GPU_0 \rightarrow GPU_1), (M, GPU_0 \rightarrow GPU_1)\}$  and  $requestString = \{(I, GPU_0), (O, GPU_1)\}$ . The first instance of the request  $(M, GPU_0 \rightarrow GPU_1)$  in *stack* is popped on line 3, and the procedure finds that request  $(I, GPU_0)$  indeed blocks the migration request. We recurse on line 11. On this call,  $stack = \{(I, GPU_0), (M, GPU_0 \rightarrow GPU_1)\}$  and  $requestString = \{(O, GPU_1)\}$ . Request  $(I, GPU_0)$  is popped from *stack*. At line 5, the condition evaluates to *false*, since requests  $(I, GPU_0)$  and  $(O, GPU_1)$  do not contend for the same CEs. Consequently, the procedure executes line 14, since *stack* is not empty (it still holds request  $(M, GPU_0 \rightarrow GPU_1)$ ). The procedure recurses once again. Now,  $stack = \{(M, GPU_0 \rightarrow GPU_1)\}$  and  $requestString = \{(O, GPU_1)\}$ . At line 5, the procedure determines that request  $(O, GPU_1)$  blocks  $(M, GPU_0 \rightarrow GPU_1)$ , since these requests contend for the same outbound CE of  $GPU_1$ . As the recursive calls to COMPUTEBLOCKINGCHAIN unwind, the blocking chain  $\{(I, GPU_0), (O, GPU_1)\}$  is constructed for initial request  $(M, GPU_0 \rightarrow GPU_1)$ .

The runtime complexity of an initial call to COMPUTEBLOCKINGCHAIN (*e.g.*, in insets (b), (c), and (d) of Figure 4.12) is a function of the request issued by job  $J_i$  and the length of the input request string. Each scenario we evaluate contains  $\rho \cdot g - 1$  requests, so each corresponding input request string is  $\rho \cdot g - 1$  requests in length. Every request processed by COMPUTEBLOCKINGCHAIN, including the request issued by job  $J_i$ , appears on *stack* at most twice (*e.g.*, when it is a migration request). Therefore, the stack may contain up to  $2 \cdot \rho \cdot g$  requests. COMPUTEBLOCKINGCHAIN processes one request from *stack* each time it is called. Thus, we may upper-bound the number of calls to COMPUTEBLOCKINGCHAIN by  $2 \cdot \rho \cdot g$ . Since the procedure contains no loops, and all comparison, stack, and list concatenation operations are  $\mathcal{O}(1)$  in complexity, the runtime complexity of COMPUTEBLOCKINGCHAIN is  $\mathcal{O}(\rho \cdot g)$ .

The insightful reader may have noticed that COMPUTEBLOCKINGCHAIN does not always find the longest blocking chain for a given scenario. Let us return to our prior scenario, with the addition of a null request in *requestString*. Suppose COMPUTEBLOCKINGCHAIN is called with  $stack = \{(M, GPU_0 \rightarrow GPU_1), (M, GPU_0 \rightarrow GPU_1)\}$  and  $requestString = \{(I, GPU_0), (N, -), (O, GPU_1)\}$ . The null request in *requestString* causes the procedure to terminate “early” and return the blocking chain  $\{(I, GPU_0)\}$  for the migration request. Since a null request can never interfere with another request, one may suppose we should eliminate all null requests from *requestString* prior to processing. If this preprocessing step were taken, then COM-



PUTEBLOCKINGCHAIN would return the chain  $\{(I, \text{GPU}_0), (O, \text{GPU}_1)\}$  for the initial request  $(M, \text{GPU}_0 \rightarrow \text{GPU}_1)$ . However, recall that our brute-force approach for finding *all* possible blocking chains evaluates all permutations of request types and request orders. Although COMPUTEBLOCKINGCHAIN returns the chain  $\{(I, \text{GPU}_0)\}$  for request  $(M, \text{GPU}_0 \rightarrow \text{GPU}_1)$  with the request string  $\{(I, \text{GPU}_0), (N, -), (O, \text{GPU}_1)\}$ , at some point, we also evaluate the request string  $\{(I, \text{GPU}_0), (O, \text{GPU}_1), (N, -)\}$  for the same migration request. With the null request at the end of the request string, COMPUTEBLOCKINGCHAIN returns the chain  $\{(I, \text{GPU}_0), (O, \text{GPU}_1)\}$ . This brute-force approach also handles permuted request strings such as  $\{(I, \text{GPU}_0), (O, \text{GPU}_1), (I, \text{GPU}_0), (O, \text{GPU}_1)\}$  and  $\{(I, \text{GPU}_0), (I, \text{GPU}_0), (O, \text{GPU}_1), (O, \text{GPU}_1)\}$ . For a migration request  $(M, \text{GPU}_0 \rightarrow \text{GPU}_1)$ , COMPUTEBLOCKINGCHAIN returns the blocking chains  $\{(I, \text{GPU}_0), (O, \text{GPU}_1)\}$  and  $\{(I, \text{GPU}_0), (I, \text{GPU}_0), (O, \text{GPU}_1), (O, \text{GPU}_1)\}$  for the respective request permutations.

**Blocking analysis with blocking chains.** We now discuss how we incorporate blocking chains, which have been computed offline, into schedulability tests. Every CE request issued by job  $J_i$  may be blocked by any one of the possible blocking chains, until all requests  $R_j \in \mathcal{R}_i^{CE}$  have been counted. We compute a tighter bound on  $b_i^{CE}$  by finding the maximal sum of all possible chains that may block job  $J_i$  across all CE requests issued by  $J_i$ . We can compute this sum using an ILP, given  $N_i^I, N_i^O, N_i^S, \mathcal{R}_i^{CE}$ , a table of all representative blocking chains, and assumed overhead costs. Of course, this is an undesirable solution since solving an ILP is NP-hard in the strong sense. Can it be avoided? Does a greedy polynomial-time algorithm exist? In general, the answer is negative. Consider the following case. Suppose job  $J_i$  makes two inbound requests to copy data to a GPU, so  $N_i^I = 2$ . Assume that job  $J_i$  has no state to migrate ( $N_i^S = 0$ ), but another job  $J_j$  does ( $N_j^S \neq 0$ ). Further suppose  $\mathcal{R}_i^{CE}$  is made up of six outbound requests and two migration requests. That is,  $\mathcal{R}_i^{CE} = \hat{\mathcal{R}}_i^{CE,0} = \{O, O, O, O, O, O, M, M\}$ . Finally, suppose we consider a simple platform where  $g = 2$  and  $\rho = 3$ . Table 4.2 depicts all representative blocking chains for such a platform with associated costs. Which two representative blocking chains, one for each request issued by job  $J_i$ , can we construct from the requests in  $\mathcal{R}_i^{CE}$  that maximizes the total blocking cost for job  $J_i$ ?

Under a greedy approach, we select the most costly chain that can be constructed from the available requests in  $\hat{\mathcal{R}}_i^{CE,k}$  for each of the  $k$  requests issued by job  $J_i$ . We continue to select the most costly chains until a chain has been selected for each of the  $k$  requests, or until the pool of interfering requests has been exhausted (*i.e.*,  $\hat{\mathcal{R}}_i^{CE,k} = \emptyset$ ). Returning to our example, we examine the “inbound” column of Table 4.2 for a list of possible chains to select for job  $J_i$ ’s two inbound requests. We first select the chain “OOOMM,” since

Parameter	Description
$\mathcal{R}_i^O$	set of outbound requests that may interfere with a request of job $J_i$
$\mathcal{R}_i^I$	set of inbound requests that may interfere with a request of job $J_i$
$\mathcal{R}_i^{P2P}$	set of migration requests that may interfere with a request of job $J_i$
$\mathcal{C}^{out}$	set of all representative outbound blocking chains for a given GPUSync configuration
$\mathcal{C}^{in}$	set of all representative inbound blocking chains for a given GPUSync configuration
$\mathcal{C}^{mig}$	set of all representative migration blocking chains for a given GPUSync configuration
$X_j^{out}$	cost of the $j^{\text{th}}$ chain in $\mathcal{C}^{out}$
$X_k^{in}$	cost of the $k^{\text{th}}$ chain in $\mathcal{C}^{in}$
$X_\ell^{mig}$	cost of the $\ell^{\text{th}}$ chain in $\mathcal{C}^{mig}$
$O_j^{out}$	number of outbound requests in the $j^{\text{th}}$ chain in $\mathcal{C}^{out}$
$O_j^{in}$	number of inbound requests in the $j^{\text{th}}$ chain in $\mathcal{C}^{out}$
$O_j^{mig}$	number of migration requests in the $j^{\text{th}}$ chain in $\mathcal{C}^{out}$
$I_k^{out}$	number of outbound requests in the $k^{\text{th}}$ chain in $\mathcal{C}^{in}$
$I_k^{in}$	number of inbound requests in the $k^{\text{th}}$ chain in $\mathcal{C}^{in}$
$I_k^{mig}$	number of migration requests in the $k^{\text{th}}$ chain in $\mathcal{C}^{in}$
$M_\ell^{out}$	number of outbound requests in the $\ell^{\text{th}}$ chain in $\mathcal{C}^{mig}$
$M_\ell^{in}$	number of inbound requests in the $\ell^{\text{th}}$ chain in $\mathcal{C}^{mig}$
$M_\ell^{mig}$	number of migration requests in the $\ell^{\text{th}}$ chain in $\mathcal{C}^{mig}$
$N_i^O$	number of outbound requests issued by job $J_i$
$N_i^I$	number of inbound requests issued by job $J_i$
$N_i^S$	number of migration requests issued by job $J_i$
$c_j^{out}$	number of instances of the $j^{\text{th}}$ chain in $\mathcal{C}^{out}$ that blocks job $J_i$
$c_k^{in}$	number of instances of the $k^{\text{th}}$ chain in $\mathcal{C}^{in}$ that blocks job $J_i$
$c_\ell^{mig}$	number of instances of the $\ell^{\text{th}}$ chain in $\mathcal{C}^{mig}$ that blocks job $J_i$

Table 4.4: Summary ILP parameters.

it is the greatest-cost chain that we can construct from the requests in  $\hat{\mathcal{R}}_i^{CE,0}$ .  $\hat{\mathcal{R}}_i^{CE,1} = \{O, O, O\}$ . No chain listed in the inbound column of Table 4.2 can be constructed from the requests in  $\hat{\mathcal{R}}_i^{CE,1}$  for job  $J_i$ 's second request. The total cost blocking cost for job  $J_i$  is 3,994 $\mu$ s (the cost of the chain “OOOMM”). What if we had made a non-greedy choice for  $J_i$ 's first request? Under a non-greedy approach, suppose we select “OOOM” for the first chain instead of “OOOMM.” Now  $\hat{\mathcal{R}}_i^{CE,1} = \{O, O, O, M\}$ . We may select the chain “OOOM” once again for the second chain. The total cost of these chains is 3,818 $\mu$ s + 3,818 $\mu$ s = 7,636 $\mu$ s—greater than the greedy approach's “bound.” This example demonstrates that the greedy approach *fails* to provide valid upper-bounds on blocking costs. The fault in the approach is that it may prematurely exhausted  $\hat{\mathcal{R}}_i^{CE,k}$  of migration requests. This prevents the construction of subsequent chains that require them. We now describe the ILP, solved once for each  $T_i \in \mathcal{T}^{gpu}$ , that we use to obtain correct bounds.

We begin by defining the variables of our ILP, which are summarized in Table 4.4. Let  $\mathcal{C}^{out}$  denote the set of all representative outbound blocking chains for a given GPUSync configuration defined by  $\rho$  and  $g$ . For example,  $\mathcal{C}^{out}$  is the set of outbound blocking chains in Table 4.2 when  $\rho = 3$  and  $g = 2$ . Similarly, let  $\mathcal{C}^{in}$

and  $\mathcal{C}^{mig}$  denote the set of all representative inbound and migration blocking chains for the same GPUSync configuration, respectively. We count the number of instances of the  $j^{th}$  outbound blocking chain in  $\mathcal{C}^{out}$  that interfere with inbound requests of job  $J_i$  with the integer variable  $c_j^{out}$ . We count the number of instances of the  $k^{th}$  inbound blocking chain in  $\mathcal{C}^{in}$  that interfere with inbound requests of job  $J_i$  with the integer variable  $c_k^{in}$ . Similarly, we count the number of instances of the  $\ell^{th}$  migration blocking chain in  $\mathcal{C}^{mig}$  that interfere with migration requests of job  $J_i$  with the integer variable  $c_\ell^{mig}$ . We now define the constants that appear in our ILP. We count the number of outbound, inbound, and migration requests that appear in the  $j^{th}$  outbound blocking chain with the constants  $O_j^{out}$ ,  $I_j^{out}$ , and  $M_j^{out}$ , respectively. For example, for the chain “IIOOM,”  $I_0^{out} = 2$ ,  $O_0^{out} = 2$ , and  $M_0^{out} = 1$ , where each value corresponds to the number of inbound, outbound, and migration requests that appear in the chain. We define similar constants,  $O_k^{in}$ ,  $I_k^{in}$ , and  $M_k^{in}$  for the  $k^{th}$  inbound blocking chain, as well as  $O_\ell^{mig}$ ,  $I_\ell^{mig}$ , and  $M_\ell^{mig}$  for the  $\ell^{th}$  migration blocking chain. Several additional constants are derived from the parameters of job  $J_i$  and the set of CE requests that may interfere with a CE request of  $J_i$ . The number of outbound, inbound, and migration requests issued by job  $J_i$  is bounded by  $N_i^O$ ,  $N_i^I$ , and  $N_i^S$ , respectively. We denote the total number of inbound, outbound, and migration requests that may interfere with *any* CE request of job  $J_i$  by  $|\mathcal{R}_i^I|$ ,  $|\mathcal{R}_i^O|$ , and  $|\mathcal{R}_i^{P2P}|$ , respectively. We now derive the coefficients that we use in the objective function of our ILP. Each coefficient denotes the cost of a single instance of a blocking chain. The coefficient  $X_j^{out}$  represents the cost of the  $j^{th}$  outbound blocking chain. We compute the value of  $X_j^{out}$  from our empirical measurements and types of requests that make up each blocking chain. More precisely,

$$X_j^{out} \triangleq X^O \cdot O_j^{out} + X^I \cdot I_j^{out} + X^{P2P} \cdot M_j^{out}. \quad (4.20)$$

Similarly,  $X_k^{in}$  represents the cost of the  $k^{th}$  inbound blocking chain. The value of  $X_k^{in}$  is computed by the equation

$$X_k^{in} \triangleq X^O \cdot O_k^{in} + X^I \cdot I_k^{in} + X^{P2P} \cdot M_k^{in}. \quad (4.21)$$

Finally,  $X_\ell^{mig}$  represents the cost of the  $\ell^{th}$  migration blocking chain. The value of  $X_\ell^{mig}$  is computed by the equation

$$X_\ell^{mig} \triangleq X^O \cdot O_\ell^{mig} + X^I \cdot I_\ell^{mig} + X^{P2P} \cdot M_\ell^{mig}. \quad (4.22)$$

With the above variables, constants, and coefficients defined, we now present the ILP we use to bound the pi-blocking any job of task  $T_i$  may experience when issuing copy engine requests. We take the solution of this ILP as the value of  $b_i^{CE}$ .

$$\text{Maximize: } \sum_{j=0}^{|\mathcal{C}^{out}|-1} X_j^{out} \cdot c_j^{out} + \sum_{k=0}^{|\mathcal{C}^{in}|-1} X_k^{in} \cdot c_k^{in} + \sum_{\ell=0}^{|\mathcal{C}^{mig}|-1} X_\ell^{mig} \cdot c_\ell^{mig} \quad (4.23)$$

$$\text{Subject to: } \sum_{j=0}^{|\mathcal{C}^{out}|-1} c_j^{out} \leq N_i^O, \quad (4.24)$$

$$\sum_{k=0}^{|\mathcal{C}^{in}|-1} c_k^{in} \leq N_i^I, \quad (4.25)$$

$$\sum_{\ell=0}^{|\mathcal{C}^{mig}|-1} c_\ell^{mig} \leq N_i^S, \quad (4.26)$$

$$\sum_{j=0}^{|\mathcal{C}^{out}|-1} I_j^{out} \cdot c_j^{out} + \sum_{k=0}^{|\mathcal{C}^{in}|-1} I_k^{in} \cdot c_k^{in} + \sum_{\ell=0}^{|\mathcal{C}^{mig}|-1} I_\ell^{mig} \cdot c_\ell^{mig} \leq |\mathcal{R}_i^O|, \quad (4.27)$$

$$\sum_{j=0}^{|\mathcal{C}^{out}|-1} O_j^{out} \cdot c_j^{out} + \sum_{k=0}^{|\mathcal{C}^{in}|-1} O_k^{in} \cdot c_k^{in} + \sum_{\ell=0}^{|\mathcal{C}^{mig}|-1} O_\ell^{mig} \cdot c_\ell^{mig} \leq |\mathcal{R}_i^I|, \quad (4.28)$$

$$\sum_{j=0}^{|\mathcal{C}^{out}|-1} M_j^{out} \cdot c_j^{out} + \sum_{k=0}^{|\mathcal{C}^{in}|-1} M_k^{in} \cdot c_k^{in} + \sum_{\ell=0}^{|\mathcal{C}^{mig}|-1} M_\ell^{mig} \cdot c_\ell^{mig} \leq |\mathcal{R}_i^{P2P}|. \quad (4.29)$$

Inequality (4.24) constrains the total number of outbound chains that may interfere with job  $J_i$  to the number of outbound CE requests issued by  $J_i$ . Similarly, Inequalities (4.25) and (4.26) constrain the total number of inbound and migration chains that may interfere with job  $J_i$  to the number of inbound and migration CE requests issued by  $J_i$ , respectively. Inequalities (4.27), (4.28), and (4.29) constrain the number CE requests that are used to compose blocking chains, by type, to the maximum number of such requests that may interfere with job  $J_i$ . For example, suppose no task in  $\mathcal{T}$  maintains state on a GPU. In this case,  $|\mathcal{R}_i^{P2P}| = 0$ , since no job ever issues a migration request. Under these conditions, the constraint represented by Inequality (4.29) ensures that no chain that contains an “M” is ever included in the calculation of  $b_i^{CE}$ .

We make a noteworthy compromise in the above ILP. Observe that the cost of each blocking chain is computed in terms integral numbers of  $X^I$ ,  $X^O$ ,  $X^{P2P}$ . These values represent the cost of transmitting a chunk of data. For example, the costs reflected in Table 4.2 are derived from the costs for 1MB chunks. What happens if a job transmits only a fractional chunk of data? The ILP will compute a pessimistic bound for  $b_i^{CE}$ . Although it may be possible to extend the above ILP to account for fractional chunks, such a

program is likely to become exceedingly complicated. Alternatively, one may assume smaller chunk sizes to reduce pessimism. This approach increases the number of CE requests issued by each task, increasing the computational complexity of the ILP. Moreover, smaller chunk sizes incur greater overheads due to DMA setup costs (recall Observation 5 from Section 4.3.1.2). In large scale schedulability experiments, such as the ones we present later in Section 4.3.6, we find that our ILP already borders on the edge of intractability within the bounds of the resources available to us on our university compute cluster, so we accept the limitations of the ILP. For smaller scale experiments, or for a system designer attempting to provision or validate a handful of task sets, more complex accounting methods may be considered feasible.

In contrast to the ILP, we can trivially support fractional chunk sizes in the request interference function,  $txif(T_i, T_j, \ell_k)$  (Equation (4.13)). As such, the constructed set  $\mathcal{R}_i^{CE}$  may incorporate fractional costs. For task sets that transmit little data, bounds computed by Equation (4.19) may actually produce *less pessimistic* bounds than our ILP.

**Detailed analysis for copy engine lock requests with system memory migration.** Detailed analysis of pi-blocking for CEs under system memory migration is easier to conceptualize and compute since there can be no transitive pi-blocking. We present detailed bounds for GPUs with two CEs first. As before, we denote pi-blocking job  $J_i$  experiences due to its CE requests with the term  $b_i^{CE}$ . We *redefine* the terms  $\mathcal{R}_i^I$ ,  $\mathcal{R}_i^O$ , and  $\mathcal{R}_i^{CE}$  as needed.

We compute  $b_i^{CE}$  in two parts:  $b_i^{CE_I}$  and  $b_i^{CE_O}$ . Let  $\mathcal{R}_i^I$  denote the sorted set of requests for copying kernel input data and task state to a GPU that may interfere with a similar request of job  $J_i$ :  $\mathcal{R}_i^I \triangleq txif(T_i, \ell^I)$ . The total worst-case pi-blocking experienced by  $J_i$  while waiting to receive the requested CE lock when system memory migrations are used with dual-CE GPUs is bounded by

$$b_i^{CE_I} = \sum_{R_j \in top((p-1) \cdot (N_i^I + N_i^S), \mathcal{R}_i^I)} L_j, \quad (4.30)$$

where  $L_j$  is equal to the length of the associated CE operation (*e.g.*,  $X^I$ ).

Let  $\mathcal{R}_i^O$  denote the sorted set of requests for copying kernel output data and task state from a GPU that may interfere with a similar request of job  $J_i$ :  $\mathcal{R}_i^O \triangleq txif(T_i, \ell^O)$ . The total worst-case pi-blocking experienced by  $J_i$  while waiting to receive the requested CE lock when system memory migrations are used with dual-CE

GPUs is bounded by

$$b_i^{CEo} = \sum_{R_j \in \text{top}((\rho-1) \cdot (N_i^O + N_i^S), \mathcal{R}_i^O)} L_j, \quad (4.31)$$

where  $L_j$  is equal to the length of the associated CE operation (*e.g.*,  $X^O$ ).

By construction, the total worst-case pi-blocking experienced by  $J_i$  while waiting for a CE lock when system memory migrations are used with dual-CE GPUs is bounded by

$$b_i^{CE} = b_i^{CEI} + b_i^{CEo}. \quad (4.32)$$

Unlike the more complicated analysis for bounds when peer-to-peer migrations are used, bounds for inbound and outbound CEs are completely isolated from one another.

For GPUs with a single CE, we combine input, output, and state operations to compute  $b_i^{CE}$  jointly. In this case, let  $\mathcal{R}_i^{CE} \triangleq \mathcal{R}_i^I \cup \mathcal{R}_i^O$ , denoting the sorted set of interfering requests for the CE of any single GPU. The total worst-case pi-blocking experienced by  $J_i$  while waiting to receive the requested CE lock when system memory migrations are used with GPUs with a single CE is bounded by

$$b_i^{CE} = \sum_{R_j \in \text{top}((\rho-1) \cdot (N_i^I + N_i^O + 2 \cdot N_i^S), \mathcal{R}_i^{CE})} L_j, \quad (4.33)$$

where  $L_j$  is equal to the length of the associated CE operation (*e.g.*,  $X^I$  or  $X^O$ ). Migration operations are counted twice, since state data is copied twice: once to system memory and once to GPU memory.

**Obtaining tighter bounds in special cases.** Before concluding this section, we discuss special cases where we can improve our engine lock blocking analysis. The analysis presented thus far has assumed a worst-case scenario where all of the  $\rho \cdot g$  tokens are held at once, since this results in maximum engine lock contention. However, there are two conditions where this degree of contention is impossible. The first condition is straightforward: if the number of tasks in  $\mathcal{T}_a^{gpu}$  is less than  $\rho \cdot g$ , *i.e.*,  $|\mathcal{T}_a^{gpu}| < \rho \cdot g$ , then at most  $|\mathcal{T}_a^{gpu}|$  tokens may be held at a given time instant. The second condition is more nuanced. If the GPU Allocator is configured like the CK-OMLP, then Rule C1 (*i.e.*, the donation-at-job-release rule) may limit the number simultaneously held tokens. Specifically, Rule C1 ensures that at most  $\hat{c}_a$  jobs may hold tokens at a time, where  $\hat{c}_a$  denotes the number of CPUs where the tasks in  $\mathcal{T}_a^{gpu}$  may execute.

We integrate these special cases into our engine lock blocking analysis. Let the variable  $Z$  take a boolean value indicating whether the GPU Allocator is configured as the CK-OMLP ( $Z = 1$ ) or not ( $Z = 0$ ). We make the following substitutions our blocking calculations. First, we substitute all instances of “ $(\rho - 1)$ ” with “ $(\min(\rho, |\mathcal{T}_a^{gpu}|, Z \cdot \hat{c}_a + \neg Z \cdot \rho) - 1)$ ” in Equations (4.10), (4.18), (4.30), (4.31), and (4.33). Also, we substitute all instances of “ $(\rho \cdot g - 1)$ ” with “ $(\min(\rho, |\mathcal{T}_a^{gpu}|, Z \cdot \hat{c}_a + \neg Z \cdot \rho) \cdot g - 1)$ ” in Equations (4.8), (4.9), (4.16), and (4.17). Finally, for detailed blocking analysis of copy engine requests when peer-to-peer migrations are used, we further constrain the objective function expressed by Equation (4.23) to prevent the inclusion of representative blocking chains greater than  $(\min(\rho, |\mathcal{T}_a^{gpu}|, Z \cdot \hat{c}_a + \neg Z \cdot \rho) \cdot g - 1)$  requests in length.

This concludes our detailed blocking analysis for engine locks.

#### 4.3.4.4 Detailed Blocking Analysis for the GPU Allocator

In this section, we discuss detailed pi-blocking analysis for the GPU Allocator. After we have performed engine lock pi-blocking analysis, we use Equation (4.5) to bound the token critical section length,  $L_i^K$ , of each  $T_i \in \mathcal{T}^{gpu}$ . We may use these bounds in the coarse-grain blocking analysis that we discussed in Section 2.1.7.2 or 2.1.7.3, provided that the GPU Allocator is configured as the R<sup>2</sup>DGLP or CK-OMLP, respectively. However, detailed blocking analysis for the GPU Allocator will yield better schedulability results. We begin by presenting detailed analysis for the GPU Allocator when it is configured as the R<sup>2</sup>DGLP. We then examine the case where the GPU Allocator is configured as the CK-OMLP.

**Detailed analysis for token request pi-blocking under the R<sup>2</sup>DGLP.** Let  $|\mathcal{T}_a^{gpu}|$  denote the number of GPU-using tasks in the  $a^{th}$  GPU cluster, which is managed by a single instance of the R<sup>2</sup>DGLP. Let  $v_i$  denote the maximum number of token critical sections that may block a token request of task  $T_i \in \mathcal{T}_a^{gpu}$ . As described by Ward *et al.* (2012),  $v_i$  depends upon the number of tokens, task that request tokens, and CPU and GPU cluster sizes. Given these parameters,  $v_i$  may be bound using one of the equations for each of the three following cases:

1. If  $|\mathcal{T}_a^{gpu}| \leq \rho \cdot g$ , then token requests are trivially satisfied, since there is always an available token for any requesting task:

$$v_i = 0. \tag{4.34}$$

2. If  $\rho \cdot g < |\mathcal{T}_a^{gpu}| \leq c$ , then no request ever overflows into the R<sup>2</sup>DGLP's PQ or DQ (every request enters an FQ immediately), so

$$v_i = \left\lfloor \frac{|\mathcal{T}_a^{gpu}| - 1}{\rho \cdot g} \right\rfloor. \quad (4.35)$$

Equation (4.35) assumes that the GPU Allocator simply load-balances the number of pending requests among FIFO queues  $\{\text{FQ}_0, \dots, \text{FQ}_{\rho \cdot g - 1}\}$ . The GPU Allocator behaves in this manner if heuristics are disabled. However, if the GPU Allocator employs heuristics to reduce migration overheads, then the number of requests in each FQ may become unbalanced—some queues are full while others are empty. We may account for this by pessimistically assuming a bound on  $v_i$  that is based upon maximum FQ length:

$$v_i = \min \left( \left\lceil \frac{c}{\rho \cdot g} \right\rceil, |\mathcal{T}_a^{gpu}| \right) - 1. \quad (4.36)$$

If a task set is schedulable when Equation (4.35) is used in analysis, but not schedulable when Equation (4.36) is used, the system designer may reduce maximum FQ length from  $\lceil c/(\rho \cdot g) \rceil$  to as little as  $\lceil |\mathcal{T}_a^{gpu}| / (\rho \cdot g) \rceil$ .<sup>12</sup> We may make this alteration to the GPU Allocator without affecting optimality, since  $|\mathcal{T}_a^{gpu}| \leq c$ . When the maximum FQ length is constrained to  $\lceil |\mathcal{T}_a^{gpu}| / (\rho \cdot g) \rceil$ , Equation (4.35) holds, even if the GPU Allocator employs heuristics. The choice to reduce maximum FQ length reflects a tradeoff between analytical schedulability and potential benefits in average-case runtime performance.

3. If  $|\mathcal{T}_a^{gpu}| > c$ , then

$$v_i = 2 \cdot \left\lceil \frac{c}{\rho \cdot g} \right\rceil - 1. \quad (4.37)$$

Equation (4.37) reflects the bound given by Equation (2.34) of Section 2.1.7.2.

Once  $v_i$  has been determined for each task, we may compute  $L_i^K$ . Let  $\mathcal{R}_i^K$  denote the sorted set of token requests that may interfere with a token request of a job of  $T_i$ . When every task  $T_i \in \mathcal{T}^{gpu}$  requests a token at most once per job, then  $\mathcal{R}_i^K \triangleq \text{txif}(T_i, \ell^K)$ , where  $\ell^K$  denotes the requested token. The total worst-case pi-blocking experienced by job  $J_i$  while waiting for a token is given by

$$b_i^K = \sum_{R_j \in \text{top}(v_i, \mathcal{R}_i^K)} L_j, \quad (4.38)$$

---

<sup>12</sup>Our implementation of GPUSync supports this degree of configurability.



where  $L_j$  denotes a bound on the token critical section length of task  $T_j$ .

**Detailed analysis for token request pi-blocking under the CK-OMLP.** We use the detailed analysis provided by Brandenburg (2011b) for the CK-OMLP in our analysis of the GPU Allocator when it is configured as the CK-OMLP. Recall from Section 2.1.7.3 that jobs experience both direct and indirect pi-blocking under the CK-OMLP. Only jobs of tasks  $T_i \in \mathcal{T}^{gpu}$  may experience direct pi-blocking. All jobs may experience indirect pi-blocking.

We bound direct pi-blocking under the CK-OMLP with the following equations:

$$\mathcal{R}_{i,a}^K \triangleq \begin{cases} \text{top}(c, \bigcup_{T_j \in \mathcal{T}_a} \text{xif}(T_i, T_j, \ell^K)) & \text{if } T_i \notin \mathcal{T}_a \\ \text{top}(c-1, \bigcup_{T_j \in \mathcal{T}_a \setminus \{T_i\}} \text{xif}(T_i, T_j, \ell^K)) & \text{if } T_i \in \mathcal{T}_a, \end{cases} \quad (4.39)$$

$$\mathcal{R}_i^K \triangleq \bigcup_{a=0}^{m/c-1} \mathcal{R}_{i,a}^K, \quad (4.40)$$

$$b_i^{K_{\text{direct}}} = \sum_{R_j \in \text{top}(\lceil \frac{c}{\rho \cdot g} \rceil - 1, \mathcal{R}_i^K)} L_j. \quad (4.41)$$

Equation (4.39) computes the set of token requests, in the worst-case, of each CPU cluster that may directly interfere with a request of  $T_i$ . This set of per-cluster requests is denoted by  $\mathcal{R}_{i,a}^K$ . Under Rule C1 (*i.e.*, the donation-at-job-release rule) of the CK-OMLP, the number of such requests is bound by  $c$  for remote clusters ( $T_i \notin \mathcal{T}_a$ ) and  $(c-1)$  for the local cluster ( $T_i \in \mathcal{T}_a$ ). This is represented by the two cases in Equation (4.39).  $\mathcal{R}_{i,a}^K = \emptyset$  if the tasks  $T_j \in \mathcal{T}_a$  do not share a GPU cluster with tasks  $T_i \in \mathcal{T}_b$ . Equation (4.40) combines the per-cluster sets of interfering requests into one,  $\mathcal{R}_i^K$ . Finally, Equation (4.41) computes a bound on direct pi-blocking experienced by task  $T_i$ , by summing the  $(\lceil c/(\rho \cdot g) \rceil - 1)$ -longest token critical sections that may directly interfere with a token request of task  $T_i$ .

We now bound indirect pi-blocking due to token requests. A job is pi-blocked indirectly while it donates its priority to another task. The time a job acts as a priority donor is bounded by the time its *donee* is blocked, plus the critical section length of the donee itself. In the worst-case, this is bounded by

$$b_i^{K_{\text{indirect}}} = \max \left( \left\{ b_j^{K_{\text{direct}}} + L_j \mid T_j \in \mathcal{T}_a \setminus \{T_i\} \right\} \right). \quad (4.42)$$

With bounds on both direct and indirect pi-blocking computed, we may bound total pi-blocking experienced by any task due to token requests with the following equation:

$$b_i^K = b_i^{K_{direct}} + b_i^{K_{indirect}}. \quad (4.43)$$

This concludes our discussion of detailed blocking analysis for GPUSync.

#### 4.3.5 Overhead Accounting

We now discuss the methods we use to integrate GPU-related overheads into overhead-aware schedulability analysis. We follow the preemption-centric interrupt accounting method that we discussed in Section 2.1.8. We use all of the formulations described therein to account for non-GPU-related overheads. In this section, we describe the enhancements we make to this analysis to account for GPU-related overheads. Specifically, those relating to GPU interrupt processing. (We account for DMA-related overheads by using the data we gathered in Section 4.3.1.2 to determine values for  $\Delta^{cpd}$ ,  $X^I$ ,  $X^O$ , and  $X^{P2P}$ .) We first discuss accounting for top-half interrupt processing. We then discuss the inflation of critical sections under GPUSync to account for locking-protocol-related self-suspensions and bottom-half interrupt processing.

Before we begin with our overhead analysis, let us define several terms and equations. We begin by rewriting the preemption-centric equation for job execution time inflation, (*i.e.*, Equation (2.42)), in the following manner:

$$\hat{e}_i \triangleq e_i + 2 \cdot (\Delta^{sch} + \Delta^{cxs}) + \Delta^{cpd} \quad (4.44)$$

$$e'_i = \frac{\hat{e}_i}{1 - u_0^{tck} - \sum_{1 \leq i \leq n} u_i^{irq}} + 2c^{pre} + \Delta^{ipi} \quad (4.45)$$

Here, in Equation (4.45), we represent the numerator of the fraction that appears in Equation (2.42) with  $\hat{e}_i$ , defined by Equation (4.44). We will incrementally inflate  $\hat{e}_i$  to account for various GPUSync overheads. We refer to values that are computed in each incremental step with a superscript, such that the superscript value indicated within the square brackets matches the labeled step that defined the inflated value (*e.g.*,  $\hat{e}_i^{[1]}$ ).

**Definition 4.5.** Let  $\hat{\mathcal{T}}_a$  denote the set of tasks that are scheduled on the CPUs of the CPU clusters that are associated with the GPU cluster of  $\mathcal{T}_a^{gpu}$ .  $\hat{\mathcal{T}}_a$  may be made up of tasks from different CPU clusters

when GPU clusters are shared among CPU clusters (*i.e.*,  $(P, G, *)$ ,  $(P, C, *)$ ,  $(P, C^{P2P}, *)$ , and  $(C, G, *)$  cluster configurations).

**Definition 4.6.** Let  $\gamma_i$  denote the maximum number of times  $T_i \in \widehat{\mathcal{T}}_a$  performs a GPU engine operation (*e.g.*, executes a kernel or transmits a chunk of data). Under the following analysis, we assume that all GPU-using tasks are configured to suspend while waiting for GPU operations to complete. Thus, each job of  $T_i$  causes at most  $\gamma_i$  interrupts to be raised by a GPU, where interrupt signals the completion of one operation.

**Definition 4.7.** Let  $\lambda$  denote the number of engine locks assigned to each GPU. The value of  $\lambda$  expresses the maximum number of simultaneous operations that may be carried out by a single GPU.<sup>13</sup>

**Definition 4.8.** Let  $\Delta^{top}$  denote the execution cost of a top-half of a single GPU interrupt.

**Definition 4.9.** Let  $\Delta^{bot}$  denote the execution cost of a bottom-half of a single GPU interrupt.

We now use the above equations and terms to account for GPU-related overheads.

#### 4.3.5.1 Accounting for Interrupt Top-Halves

We now account for top-half interrupt processing overheads. Interrupts always execute with maximum priority, so accounting for priority-inversions due to the top-halves of GPU interrupts is straightforward: we assume that a job  $J_{i,u}$  is affected (or “hit”) by every GPU top-half that may be raised while  $J_{i,u}$  executes. We assume that GPU interrupts are arbitrarily delivered to CPUs where tasks  $T_i \in \widehat{\mathcal{T}}_a$  may execute—other CPUs are shielded from processing these interrupts. We compute the total number of these interrupts with the following equation:

$$H_i = \sum_{T_j \in \widehat{\mathcal{T}}_a \setminus \{T_i\}} \gamma_j \cdot \text{tif}(T_i, T_j). \quad (4.46)$$

Next, we inflate job execution cost to place an upper bound on the burden of processing GPU interrupt top-halves:

$$\hat{e}_i^{[1]} = \hat{e}_i + H_i \cdot \Delta^{top}. \quad (4.47)$$

This bound is very pessimistic, but it is also safe.

---

<sup>13</sup>This statement does not hold if multiple kernels are allowed to execute simultaneously on an execution engine. However, GPUSync’s execution engine locks explicitly forbid this possibility.

#### 4.3.5.2 Accounting for Interrupt Bottom-Halves

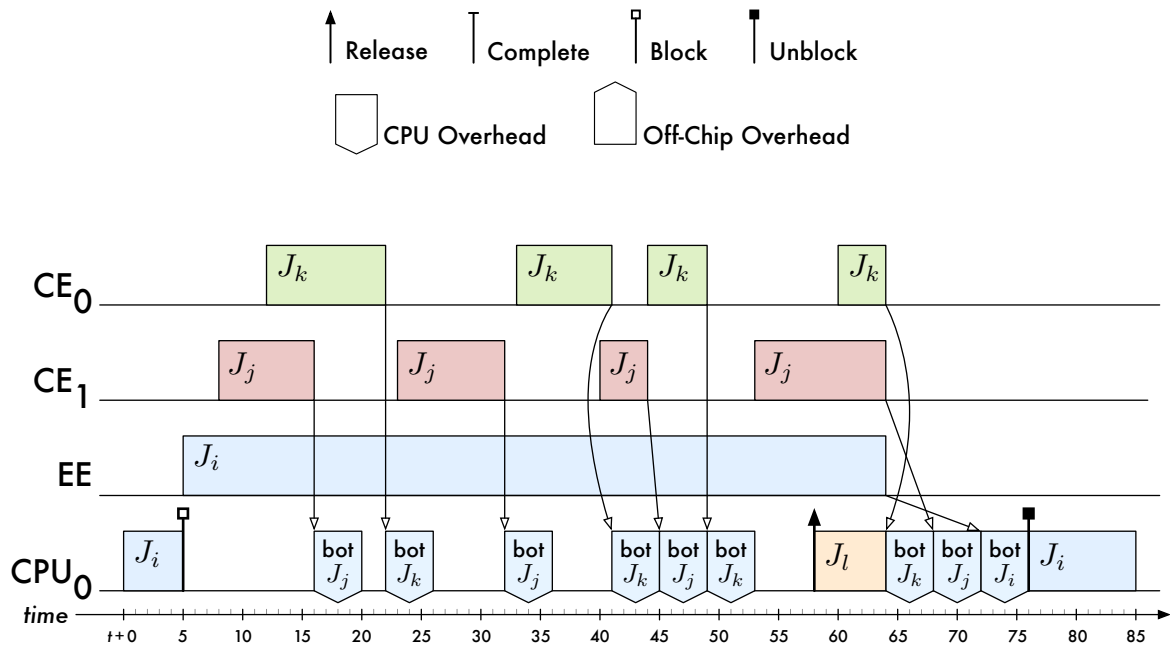
We now consider overheads associated with interrupt bottom-half processing. We account for these overheads in two ways:

1. **Inflate engine request lengths.** Bottom-half processing overheads may be incurred by a job  $J_{i,u}$  while it holds an engine lock, effectively lengthening the engine lock critical section. This may indirectly delay any job waiting to obtain the engine lock in question. This must be accounted for in pi-blocking analysis. We accomplish this by inflating engine request lengths prior to pi-blocking analysis.
2. **Inflate job WCET.** Bottom-half processing overheads may be incurred directly by a job  $J_{i,u}$ . These overheads represent the cost of OS and other system work that  $J_{i,u}$  may be required to perform.

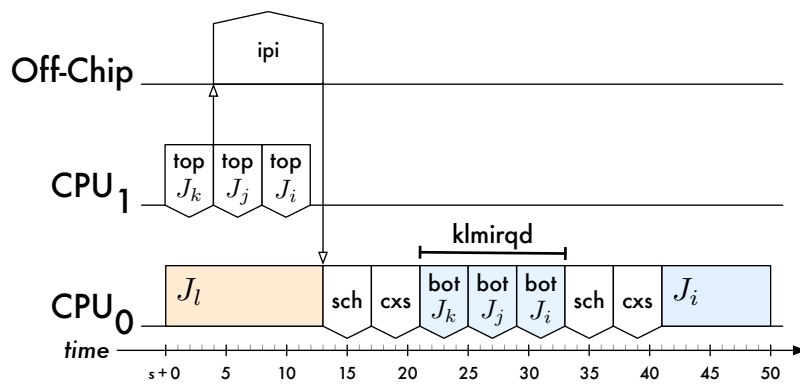
We first consider request length inflation and then job WCET inflation.

**Request length inflation.** We begin with analysis of engine request lengths. The engine request length of a job  $J_{i,u}$  can be affected by the GPU operations of other jobs assigned to the same GPU. Recall that under GPUSync, all interrupt bottom-halves of each GPU are processed serially by a per-GPU klmirqd thread. This thread inherits the maximum priority of any task that has *suspended* while waiting for a GPU operation to complete. This ensures that all priority inversions are bounded. However, this does not isolate this maximum-priority task from the GPU operations of other threads. Such a case is depicted in Figure 4.13.

In Figure 4.13(a), job  $J_i$  issues an operation to an EE within an engine critical section, and  $J_i$  suspends at time  $t + 5$ . A number of bottom-halves are processed between time  $t + 16$  and  $t + 53$  by the klmirqd thread at job  $J_i$ 's priority—we examine these later when we discuss WCET inflation. At time  $t + 64$ , three events occur simultaneously: (i) a DMA operation of job  $J_k$  on CE<sub>0</sub> completes; (ii) a DMA operation of job  $J_j$  on CE<sub>1</sub> completes; and (iii) the GPU kernel of job  $J_i$  completes. As a result, three GPU interrupts are raised in quick succession. In Figure 4.13(b), we examine what occurs next in greater detail (we change the timescale in this figure to ease presentation). Here, CPU<sub>1</sub> processes the interrupt top-halves for the interrupts raised by the GPU. The completion of the first top-half at time  $s + 4$  triggers an IPI to be sent to CPU<sub>0</sub> to wake up and schedule the associated klmirqd thread. This IPI is received at time  $s + 13$ . This triggers the scheduler on CPU<sub>0</sub> to run, and  $J_i$  is preempted. This incurs both a scheduling ( $\Delta^{sch}$ ) and a context switch ( $\Delta^{cxs}$ ) overhead. At time  $s + 21$ , the klmirqd thread begins processing pending bottom-halves. By the time the klmirqd thread begins execution, the remaining GPU top-halves have already been enqueued for processing. The klmirqd



(a) High-level view of bottom-half overhead processing.



(b) Scenario where overheads increase engine critical section length.

Figure 4.13: Schedules with overheads due to bottom-half interrupt processing.

thread executes each bottom-half, completing at time  $s + 33$ . Job  $J_i$  is unblocked once its bottom-half is processed, so  $J_i$  is scheduled next on CPU<sub>0</sub>, resulting in additional scheduling and context switch overheads.<sup>14</sup>

We assume that a system designer incorporates the top-half and bottom-half processing overheads into the provisioned execution time of job  $J_i$ 's *own* GPU kernel. That is, we do not consider job  $J_i$  to be delayed by  $\Delta_{J_i}^{top}$  or  $\Delta_{J_i}^{bot}$  (*i.e.*, the top- and bottom-half overheads of  $J_i$ ). However, we do consider job  $J_i$  to be delayed for the duration that its bottom-half is ready for processing, but is not scheduled. IPIs and the bottom-halves of other jobs cause these delays. In Figure 4.13(b), we see that the processing of  $J_i$ 's bottom-half can be delayed by up to one IPI and two scheduling and context switch operations. Also, in this particular example, job  $J_i$  is delayed by two top- and bottom-halves (those of jobs  $J_k$  and  $J_j$ ). In general, job  $J_i$  may be delayed by up to  $\min(\lambda, \rho) - 1$  top- and bottom-halves.

To account for these overheads, we inflate each engine request length,  $L_{i,u,k}$ , of job  $J_{i,u}$  using the following equation:

$$L_{i,u,k}^{[2]} \triangleq L_{i,u,k} + (\min(\lambda, \rho) - 1) \cdot (\Delta^{bot} + \Delta^{top}) + 2(\Delta^{sch} + \Delta^{csx}) + 2\Delta^{ipi}. \quad (4.48)$$

We charge for *two* IPIs in the above equation. The first IPI overhead accounts for the situation we observed in Figure 4.13(b). The second IPI charge accounts for the situation where job  $J_i$  is scheduled on a different CPU than the klmirqd thread. We may *drop* the IPI ( $\Delta^{ipi}$ ) component of Equation (4.48) when both CPUs and GPUs are partitioned (*i.e.*,  $(P, P, *)$ ), since GPU top-halves, the klmirqd thread, and job  $J_i$  are guaranteed to be scheduled on the same CPU.

Overheads due to scheduling klmirqd threads are not the only scheduling overheads to consider. Recall that in more recent version of the CUDA runtime, that user-space callback threads, one per GPU per process, are responsible for waking jobs that are suspended waiting for their GPU operation to complete. This callback thread is only active while it wakes the suspended thread, and it sleeps otherwise. GPUSync schedules the callback thread with the current priority of the corresponding job  $J_i$ , but only while  $J_i$  is suspended waiting for a GPU operation to complete. We assume the execution cost of the callback thread is already captured by the provisioning of  $L_{i,u,k}$ . However, we must include thread scheduling costs of the callback thread, since these overheads delay the waking of job  $J_i$ . System call overheads must also be considered, since the callback thread executes in user-space, and wakes up the suspended task through a system call. These overheads are

---

<sup>14</sup>In this example, we ignore the effects of any GPGPU runtime callback threads. In actuality, in GPGPU runtimes where callback threads are used, a callback thread of job  $J_i$  would be scheduled at time  $t = 41$ , not job  $J_i$  itself. We examine the effects of callback threads, shortly.

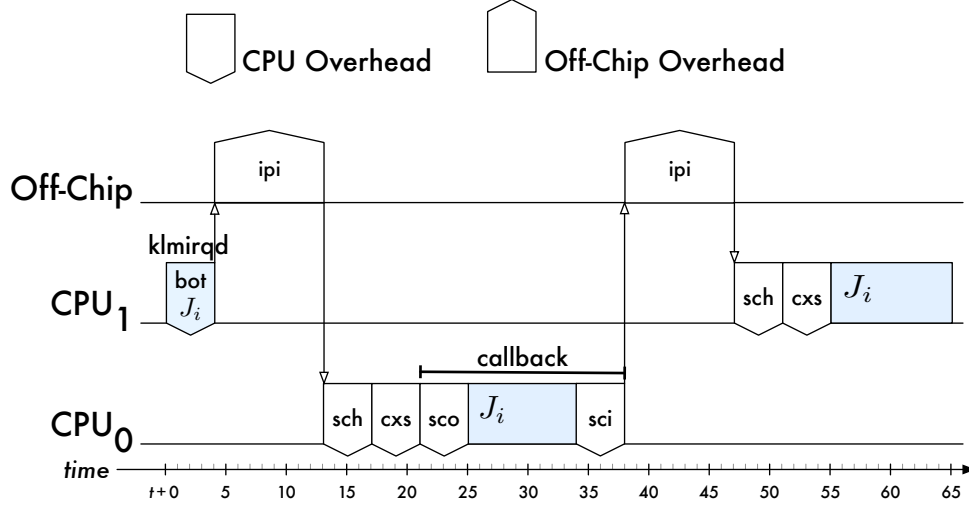


Figure 4.14: Schedule depicting callback overheads.

depicted in Figure 4.14. Figure 4.14 begins with the bottom-half processing to provide a frame of reference. The overheads prior to time  $t + 21$  have already been accounted for by Equation (4.48). At time  $t + 21$ , the callback thread returns from any prior-made system call, as indicated by the system-call-out overhead,  $\Delta^{sco}$ . Since the callback thread relinquishes any inherited priority within a system call the moment it wakes up a job, the callback thread may still be within the OS when it is scheduled at time  $t + 21$ . At time  $t + 34$ , the callback thread makes the system call to wake job  $J_i$ . This results in an IPI to schedule job  $J_i$  on CPU<sub>1</sub>—in general, we cannot guarantee that job  $J_i$  will be scheduled on the same CPU as the callback thread. Job  $J_i$  is finally scheduled at time  $t + 55$ , where it can release its engine lock, completing its engine request.

We must inflate engine request lengths to account for delays due to these callback-related overheads. We do so with the following equation:

$$L_{i,u,k}^{[3]} \triangleq L_{i,u,k}^{[2]} + \Delta^{sch} + \Delta^{cxs} + \Delta^{ipi} + \Delta^{sci}. \quad (4.49)$$

Here, we do not charge IPI, scheduling, and context switch costs to schedule the callback thread itself (*i.e.*, events prior to time  $t + 21$  in Figure 4.14), since these are already captured by Equation (4.48). However, we must still account for the overheads to wake and schedule job  $J_i$ ; we do so by inflating the engine request lengths of  $J_{i,u}$  by  $\Delta^{sch} + \Delta^{cxs} + \Delta^{ipi}$ . We also inflate the request length by system call in and out ( $\Delta^{sci}$  and  $\Delta^{sco}$ , respectively) to account for system call overheads. As with Equation (4.48), we may also drop  $\Delta^{ipi}$  from Equation (4.49) when both CPUs and GPUs are partitioned (*i.e.*,  $(P, P, *)$ ).

This completes the description of steps we use to inflate engine request lengths. *Please note that additional inflations are necessary to account for standard locking protocol overheads by applying Equation (2.46) to both engine and token lock requests.* After this has been done, bounds on pi-blocking may be computed (using the blocking analysis of Section 4.3.4.3) using the inflated request lengths.

**WCET inflation.** The incorporation of inflated engine request lengths in blocking analysis models how bottom-half processing overheads incurred by job  $J_i$  can increase the pi-blocking experienced by job  $J_v$ . We must also charge overheads caused by bottom-half and callback thread processing to  $J_i$  itself.

In Figure 4.13(a), we see several bottom-halves that are processed by a klmirqd daemon in the interval  $[t + 16, t + 52)$ . These are scheduled under job  $J_i$ 's priority. Under usual circumstances, we inflate the provisioned execution time of job  $J_i$  to cover any work performed under  $J_i$ 's priority. *However*, when these bottom-halves are scheduled on a CPU concurrently with job  $J_i$  when it is scheduled on a GPU engine, there is a “loophole” in suspension-oblivious analysis that we may exploit to cover bottom-half execution costs without needing to inflate job  $J_i$ 's provisioned execution time. Under suspension-oblivious analysis, all GPU execution time of job  $J_i$  is masked analytically by fictitious CPU demand. For example, in Figure 4.13(a), the GPU kernel of job  $J_i$  executes during the time interval  $[t + 5, t + 64)$ , for a total of 59 time units. Under suspension-oblivious analysis, job  $J_i$  is correspondingly provisioned with 59 time units of CPU execution time. This CPU budget may only be consumed while job  $J_i$  executes on a GPU, but  $J_i$  cannot execute on a CPU during this time, since it is blocked on the completion of a GPU operation. However, we may schedule *other* useful work under job  $J_i$ 's priority during this time, making use of  $J_i$ 's otherwise unusable budget. This is the loophole in suspension-oblivious analysis we exploit to cover the costs of processing bottom-halves under job  $J_i$ 's priority while  $J_i$  executes on a GPU.

Job  $J_i$  must only be provisioned with additional CPU budget to cover GPU-related overheads that the job may incur while the job is not scheduled on a GPU. We already examined these overheads in the derivations of Equations (4.48) and (4.49). We now apply them to inflate the provisioned execution time of job  $J_i$  instead of engine request lengths. We do so using the following two equations to account for bottom-half and callback processing, respectively:

$$\hat{e}_i^{[4]} = \hat{e}_i + \gamma_i \cdot (\min(\lambda, \rho) \cdot \Delta^{bot} + 2(\Delta^{sch} + \Delta^{csx}) + 2\Delta^{ipi}), \quad (4.50)$$

$$\hat{e}_i^{[5]} = \hat{e}_i^{[4]} + \gamma_i \cdot (2(\Delta^{sch} + \Delta^{csx}) + \Delta^{sci} + \Delta^{sco} + \Delta^{ipi}). \quad (4.51)$$



For each of job  $J_i$ 's of GPU operations, denoted by  $\gamma_i$ , Equation (4.50) charges the cost of processing  $\min(\lambda, \rho)$  bottom-halves (*i.e.*, the cost of processing  $(\min(\lambda, \rho) - 1)$  bottom-halves of other jobs, plus the bottom-half of job  $J_i$  itself) and the cost of scheduling the klmirqd daemon. Equation (4.51) charges the cost of scheduling the callback thread, once for each of job  $J_i$ 's GPU operations. *After Equations (4.50) and (4.51) have been used to account for GPU-related overheads, we must still apply Equation (2.45) to account for the standard locking protocol overheads, with  $\eta_i = \gamma_i + 1$ , where we add one to  $\gamma_i$  to account for a single token lock request of  $J_i$ .*

#### 4.3.5.3 Limitations

In this section, we explain and justify three pragmatic compromises we make in the analysis presented in Sections 4.3.5.1 and 4.3.5.2.

**GPU interrupt processing and cache affinity.** The observant reader may notice that we have not included cache affinity loss overheads (*e.g.*,  $\Delta^{cpd}$  or  $\Delta^{cid}$ ) in Equations (4.47), (4.48), (4.49), (4.50), or (4.51). This is intentional. We expect the cache working set size of bottom-half and callback routines to be small, since these routines merely route GPU kernel completion notifications to waiting tasks—there is no heavy data processing to be done. Nevertheless, it would be desirable to include these overheads in analysis. However, we lack a straightforward method for accurately determining the associated working set sizes of GPU top-half, bottom-half, and callback processing. Moreover, we fear that the inclusion of overly pessimistic cache affinity loss overheads in these equations would render our schedulability analysis too pessimistic to be meaningful, since the costs of these overheads would be quickly compounded in blocking analysis. As a result, the differences among different GPUSync configurations that we seek to highlight and explore would be obscured.

**Interrupt processing and engine request lengths.** In Equation (4.48), we inflated each engine request length to incorporate the bottom-half processing that may occur under  $J_i$ 's priority before the pending bottom-half of  $J_i$  is itself is processed. Similarly, Brandenburg (2011b) argues that critical sections should also be inflated to account for *every* interrupt that may occur within a critical section that is arbitrated by any locking protocol that does not disable interrupts (*e.g.*, suspension-based locking protocols). However, he also acknowledges that such an approach results in “tremendous pessimism” that may increase critical section lengths “by several orders of magnitude.” Partly motivated by a desire to explore hard real-time

schedulability, Brandenburg side-steps this issue by localizing all interrupt handling and scheduling decisions on a specially designated “release-master” CPU.

Although GPUSync does not forbid the use of a release-master CPU, we opt to not use one in this work for three reasons. First, a system CPU must be sacrificed if no real-time tasks are *partitioned* onto (*i.e.*, assigned to) the release-master CPU. This results in a compulsory CPU utilization loss of 1.0. This is something that we would rather avoid. Second, if real-time tasks are partitioned onto the release-master CPU, then these tasks experience a high-degree of interference from all system interrupts. Schedulability analysis for this release-master partition will be very pessimistic. Finally, a release-master CPU can introduce asymmetry in CPU clusters. For example, on our twelve-core evaluation platform, if we reserved one CPU as a release-master and form CPU clusters along NUMA boundaries, then one CPU cluster would have five CPUs, while the other would have six. This adds complications to task partitioning and schedulability analysis.

As with GPU interrupt processing and cache affinity loss, if we were to inflate critical sections to account for worst-case interrupt processing scenarios, the resulting schedulability analysis would be too pessimistic to hold any meaning for us. Instead, we compromise by accounting for interrupts by inflating job execution time (*e.g.*, Equation (4.47)), and we do not further inflate critical section lengths. Since our stated research goal is in discovering the best methods for supporting *soft* real-time systems with GPUs, we feel that this more relaxed model has acceptable limitations and remains sufficiently conservative to model reality.

**GPU interrupt latency.** Throughout our preemption-centric analysis, we have used the overhead  $\Delta^{ipi}$  to account for delays in interprocessor interrupt latency. This overhead captures the latency between the time when CPU<sub>*a*</sub> sends a signal (*i.e.*, an interrupt) to CPU<sub>*b*</sub>, to the time when CPU<sub>*b*</sub> actually receives said signal. There are analogous interrupt latencies between CPUs and GPUs. For example, there is a delay between the time a GPU raises an interrupt to notify the host platform of GPU operation completion, to the time a CPU actually receives said interrupt. We do not *directly* account for these overhead latencies.

We ignore GPU interrupt latency overheads because we know of no reasonable method to measure them. For CPUs, we can directly measure IPI latencies, since CPU clocks can be synchronized—we merely log a timestamp each time an IPI is sent and received, and we take the time difference as an observed IPI latency. Unfortunately, CPUs and GPUs do not share a synchronized clock, so we cannot apply the same methodology. Alternatively, we could *estimate* GPU-related interrupt latencies with the following multi-step experiment,

where: (i) a GPU signals CPU; (ii) this CPU echoes back a signal to the GPU; (iii) we measure the delay between when the GPU sends its signal to when it receives the echo; (iv) we divide this delay by two to give an estimated signal latency. Unfortunately, it is not clear to us how this experiment can actually be performed. While we may craft software that triggers these signals to be sent, this software must sit atop several layers of closed-source software. The execution time of these additional layers prevents us from making accurate measurements. Moreover, the above experiment assumes that CPU-to-GPU interrupt latency is commensurate with GPU-to-CPU interrupt latency. We do not know if this assumption is well-founded.

Given these issues, we instead account for GPU-related interrupt latencies *indirectly* by including these overheads in the provisioned GPU execution time of GPU kernels and DMA operations. Indeed, this was what was done in Section 4.3.1.2, where we characterized the cost of DMA operations. The time taken to complete each DMA operation was measured by the CPU process that issued the work. Each DMA measurement inherently includes both CPU-to-GPU and GPU-to-CPU interrupt latencies. This compromise prevents us from possibly exploiting tighter analytical methods. For example, IPI latencies are not inflated to account for CPU scheduler tick overheads in preemption-centric accounting, because the message-passing mechanisms to deliver an IPI occur “off-chip” in parallel with CPU processing (see Figure 2.12). The message-passing mechanisms that lead to latencies in GPU interrupts similarly occur off-chip. However, since we implicitly incorporate these overheads into engine request lengths, these overheads are treated as CPU execution time under suspension-oblivious analysis. Consequently, these overheads are incorporated into the numerator of Equation (4.45), where they are inflated (by way of the denominator) to account for CPU scheduler ticks, even though CPU scheduler ticks do not affect GPU interrupt latencies in reality. Ultimately, our approach is more pessimistic, albeit safe.

This concludes our accounting of GPU overheads in schedulability analysis.

### 4.3.6 Schedulability Experiments

In this section, we assess tradeoffs among the configuration options we described in Section 4.3.2 by presenting the results of overhead-aware schedulability studies. We randomly generated task sets of varying characteristics and tested them for schedulability using the methods described above. We now describe the experimental process we used.

#### 4.3.6.1 Experimental Setup

There is a wide space of system configuration and task set parameters to explore. We evaluated each of the nine high-level configurations illustrated in Figure 4.1, plus an additional three configurations where GPUs are clustered into two clusters of four GPUs apiece. As we discussed in Section 4.3.2, these configurations are not exhaustive, but we feel they are the simplest and most practical configurations for each combination of CPU and GPU cluster configurations. For instance, in  $(P, P)$  of Figure 4.1, four partitioned CPUs have no attached GPU; these CPUs may only schedule tasks of  $\mathcal{T}^{cpu}$ . Such a configuration is a natural extension of existing uniprocessor, uni-GPU methods. Each considered configuration was tested with several values of  $\rho$ :  $\rho = 1$  to examine schedulability under exclusive GPU allocation;  $\rho = 3$  to explore schedulability when all GPU engines (one EE and two CEs) are given the opportunity to operate simultaneously; and  $\rho = 2$  to see if there is a balance to strike between  $\rho = 1$  and  $\rho = 3$ . The configuration  $(*, P)$  was also tested with  $\rho = \infty$  since  $\rho$ 's role in facilitating migrations is moot.

Random task sets for schedulability experiments were generated according to several parameters in a multistep process. Task utilizations were generated using three uniform distributions:  $[0.01, 0.1]$  (*light*),  $[0.1, 0.4]$  (*medium*), and  $[0.5, 0.9]$  (*heavy*). Task periods were generated using two uniform distributions with ranges  $[33\text{ms}, 100\text{ms}]$  (*moderate*), and  $[200\text{ms}, 1000\text{ms}]$  (*long*).<sup>15</sup> Tasks were generated by selecting a utilization and period until reaching a desired task set utilization. The task set was then randomly subdivided into  $\mathcal{T}^{gpu}$  and  $\mathcal{T}^{cpu}$ . The number of tasks in  $\mathcal{T}^{gpu}$  was set to be: 33%, 66%, or 100% of the task set size. For tasks in  $\mathcal{T}^{gpu}$ , kernel execution times were generated using three uniform distributions with ranges  $[10\%, 25\%]$ ,  $[25\%, 75\%]$ , and  $[75\%, 95\%]$  of task execution time (a corresponding amount of time was subtracted from CPU execution time). For simplicity, we model each task in  $\mathcal{T}^{gpu}$  as executing one kernel per job. Each such job has one GPU critical section. Input/output data sizes were generated using three values: 256KB (*light*), 2MB (*medium*), and 8MB (*heavy*). A selected data size was evenly split between  $z_i^I$  and  $z_i^O$ . Task GPU state size was generated using three values: 0%, 25%, and 100% of  $T_i$ 's combined input/output data size. In order to keep our study tractable, all tasks were assigned a CPU cache working set size of 4KB. For tasks in  $\mathcal{T}^{gpu}$ , 5% of its CPU execution time was determined to be within the task's single GPU critical section. Overheads and data transmission times were taken from four data sets: average-case (AC) observations in an idle system

---

<sup>15</sup>These periods are inspired by the sensor streams GPUs may process. Moderate periods represent video-based sensors. Long periods model slower sensors such as LIDAR.

(AC/I); AC observations in a loaded system (AC/L); worst-case (WC) observations in an idle system (WC/I); and WC observations in a loaded system (WC/L). Due to the *extremely* long tails of the distributions of observed for GPU top-half and bottom-half interrupt execution times (depicted in Figures 4.2(a) and 4.3(a), respectively), we take the 99.9<sup>th</sup> percentiles of these measurements as our “worst-case.” We presume that extreme outlier observations are the result of software bugs in the GPU driver that could be fixed before a GPU were deployed in a serious real-time system.

A unique combination of the above system configurations and task set parameters defined a set of experimental settings, 75,816 in all. Under each set of experimental parameters, for each 0.25 increment in system utilization range (0, 12] (reflecting the range of system utilizations supported by our twelve-core test platform), we generated between 500 and 4,000 task sets.<sup>16</sup> Task sets were partitioned to the CPU/GPU clusters in three phases:

*Phase 1:*  $\mathcal{T}^{gpu}$  was partitioned among the GPU clusters, using the worst-fit heuristic in decreasing GPU utilization order, where GPU utilization was given by

$$u_i^{gpu} \triangleq \frac{q_i + e_i^{gpu} + xmit(z_i^I, z_i^O, z_i^S)}{p_i}. \quad (4.52)$$

*Phase 2:*  $\mathcal{T}^{gpu}$  was then partitioned among CPU clusters, in accordance with experimental parameters, using the worst-fit heuristic in decreasing utilization order, where task utilization was given by Equation (4.1). Bounds for pi-blocking terms were calculated and incorporated into each CPU cluster’s (estimated) total utilization.

*Phase 3:*  $\mathcal{T}^{cpu}$  was then partitioned among the CPU clusters using the worst-fit heuristic in decreasing utilization order, where task utilization was calculated using Equation (2.5) (*i.e.*, the standard definition of task utilization under the sporadic task model).

Task sets were tested for bounded response time. Task execution time and request critical section lengths were inflated to incorporate system overheads and s-oblivious pi-blocking. Tardiness bounds were computed using CVA analysis of Erickson (2014) (see Section 2.1.5). Approximately 2.8 billion task sets were tested. We used the KillDevil compute cluster at the University of North Carolina, at Chapel Hill, to perform our experiments, consuming over 85,000 CPU hours on modern Intel Xeon processors (models

---

<sup>16</sup>After testing a minimum of 500 task sets, additional task sets were generated until average schedulability fell within a three percentage-point interval with 95% confidence, or until 4,000 task sets had been tested.

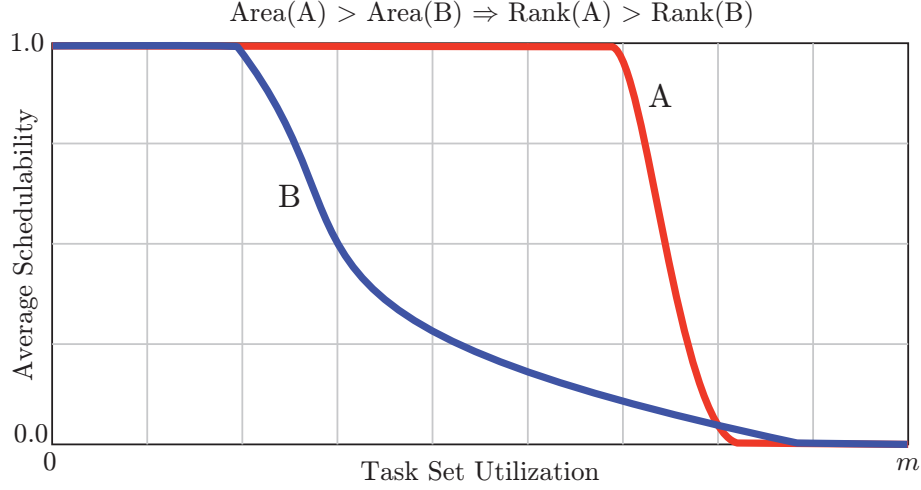


Figure 4.15: Illustrative ranking of configuration  $\mathcal{A}$  against configuration  $\mathcal{B}$ .

X5670 and E5-2670). Our experimental tools were implemented on top of the schedulability test toolkit SchedCAT (Brandenburg, 2011a). Although most of SchedCAT is implemented in the Python scripting language, performance-critical paths are implemented in C/C++. Our experimental tools used the commercial Gurobi optimization solver (Gurobi Optimization, Inc., 2015) to solve the ILP we use to bound pi-blocking for configurations with peer-to-peer migration. We point out these implementation details of our experimental tools to highlight the fact that our CPU hours were used with reasonable efficiency.

#### 4.3.6.2 Results

With over 75,000 experiments, it is infeasible to compare different system configurations by examining individual schedulability curves alone. Since our primary goal is to compare the effectiveness of each configuration, we devised the following ranking method to collapse our results into something more manageable. For every unique combination of task set parameters, we determined a “sub-rank” for each system configuration from first to last place. These sub-rankings were determined by comparing the area under each system configuration’s schedulability curve. A larger area under the curve indicates better schedulability. An illustrative example is shown in Figure 4.15. In this example with two system configurations  $\mathcal{A}$  and  $\mathcal{B}$ , configuration  $\mathcal{A}$  has a first-place sub-rank since the area under  $\mathcal{A}$ ’s curve is greater (*i.e.*, more task sets were schedulable under  $\mathcal{A}$ ). A *final* rank for each system configuration was determined by computing for each configuration, the median, average, and standard deviation of its sub-ranks. We then ranked system

configurations according to median sub-rank, tie-breaking by average sub-rank. This ranking approach was applied separately to results from each of our four overhead datasets.

Tables 4.5, 4.6, 4.7, and 4.8 show configuration rankings assuming worst-case loaded system overheads (WC/L), worst-case idle system overheads (WC/I), average-case loaded system overheads (AC/L), and average-case idle system overheads (AC/I), respectively. The columns labeled “Rank” give each configuration’s final rank. Observe that each table is sorted according to final rankings. The following columns give the median, average, and standard deviation of each configuration’s sub-ranks. Entries in the columns labeled “(CPU,GPU, $\rho$ )” identify the ranked system configuration. Here, we extend the tuple-notation from Section 4.2 to include  $\rho$ . The next three columns give the final rank of a configuration under the *other* overhead data sets. For a given row, we may compare the values of these columns against each other, and the value in the “Rank” column, to discern how a system configuration’s ranking changes under different overhead conditions. In order to fit each table onto a single page, the left-half of each table gives the rankings for the top 29 ranked configurations, and the right-half gives the rankings for the remaining configurations. We make the following observations.

**Observation 13.** *Clustered CPU scheduling with partitioned GPUs and  $\rho = \infty$  had the highest rank under three of the four overhead conditions.*

We may observe this in the first row of Table 4.5. We see that  $(C, P, \infty)$  ranked 1<sup>st</sup> under WC/L overheads. In the same row, we see that  $(C, P, \infty)$  also ranked 1<sup>st</sup> under WC/I and AC/L overheads, and ranked 2<sup>nd</sup> under AC/I overheads. Under AC/I overheads, we see that  $(G, P, \infty)$  ranks 1<sup>st</sup>. The considered overheads are lightest under AC/I assumptions, so  $(G, P, \infty)$  does well. As observed by Brandenburg (2011b), global CPU scheduling provides good soft real-time schedulability, provided that overheads are low; we see this here.

*These results show, in the context of these experiments, that partitioned GPU scheduling, when paired with clustered or global CPU scheduling, provides the best overall performance.*

**Observation 14.** *Clustered CPU scheduling with partitioned GPUs and  $\rho = \infty$  was not always the best configuration.*

We already observed that  $(G, P, \infty)$  had the highest rank under AC/I overheads. However, we can also recognize this observation in the remaining tables. In Tables 4.5, 4.6, and 4.7, compare the Median and Average sub-rank values for  $(C, P, \infty)$ . If  $(C, P, \infty)$  always had the highest rank, then Median and Average

Rankings Under Worst-Case Overheads, Loaded															
Rank	Median	Avg	$\sigma$	(CPU,GPU, $\rho$ )	WC/I	AC/I	AC/L	Rank	Median	Avg	$\sigma$	(CPU,GPU, $\rho$ )	WC/I	AC/I	AC/L
1	1	1.82	2.72	( $C,P,\infty$ )	1	2	1	30	29	28.17	8.20	( $G,C_4,3$ )	29	25	28
2	4	5.72	5.00	( $C,P,2$ )	2	5	4	31	31	27.09	8.89	( $C,C_2,1$ )	32	34	32
3	4	5.73	5.55	( $C,P,3$ )	3	3	3	32	32	32.44	6.11	( $G,C_2^{P2P},1$ )	33	31	33
4	5	10.90	10.03	( $G,P,\infty$ )	4	1	2	33	33	33.14	11.94	( $G,G,3$ )	31	27	30
5	7	9.02	8.45	( $P,P,\infty$ )	8	7	7	34	34	28.76	11.34	( $C,C_4^{P2P},1$ )	35	36	35
6	8	15.34	13.02	( $G,P,3$ )	6	4	5	35	34	34.30	6.16	( $G,C_2,1$ )	34	33	34
7	9	15.59	12.88	( $G,P,2$ )	5	6	6	36	35	29.77	12.76	( $C,C_4,1$ )	37	38	37
8	10	12.84	7.15	( $C,C_2^{P2P},2$ )	7	10	8	37	35	33.78	8.55	( $P,C_2^{P2P},1$ )	39	40	39
9	10	13.08	6.96	( $C,C_2,2$ )	9	11	9	38	37	34.30	9.34	( $P,C_2,1$ )	41	41	41
10	12	13.86	8.49	( $P,P,3$ )	12	14	14	39	37	35.29	8.07	( $G,C_4^{P2P},1$ )	36	35	36
11	12	13.89	8.95	( $P,P,2$ )	10	13	16	40	38	36.17	9.01	( $G,C_4,1$ )	38	37	38
12	13	13.91	6.04	( $C,P,1$ )	11	18	10	41	39	35.03	9.57	( $P,C_2,2$ )	43	43	42
13	15	15.06	7.22	( $C,C_2^{P2P},3$ )	14	20	17	42	39	35.37	9.46	( $P,C_2^{P2P},2$ )	42	42	43
14	15	15.64	7.14	( $C,C_2,3$ )	17	23	18	43	40	36.27	9.54	( $P,C_2,3$ )	45	45	44
15	15	16.03	6.09	( $C,C_4,2$ )	15	12	13	44	40	36.35	9.73	( $P,C_2^{P2P},3$ )	44	44	45
16	16	16.52	6.82	( $C,C_4^{P2P},2$ )	13	15	15	45	44	40.36	11.83	( $G,G,1$ )	40	39	40
17	19	18.94	7.80	( $C,C_4^{P2P},3$ )	20	29	23	46	47	45.25	6.75	( $P,C_4^{P2P},1$ )	46	46	46
18	19	19.17	7.57	( $C,C_4,3$ )	22	28	20	47	47	45.72	6.76	( $P,C_4,1$ )	47	47	47
19	19	22.35	11.16	( $G,C_2^{P2P},2$ )	16	8	11	48	48	46.26	6.94	( $P,C_4,2$ )	49	48	48
20	20	22.66	10.78	( $G,C_2,2$ )	18	9	12	49	48	46.50	6.79	( $P,C_4^{P2P},2$ )	48	49	49
21	22	18.36	10.16	( $P,P,1$ )	19	30	26	50	48	46.86	6.37	( $P,C_4,3$ )	50	50	50
22	22	23.13	9.82	( $G,P,1$ )	21	22	19	51	49	47.14	6.76	( $P,C_4^{P2P},3$ )	51	51	51
23	23	24.57	10.10	( $G,C_2^{P2P},3$ )	23	17	21	52	53	52.82	1.82	( $P,G,1$ )	52	52	53
24	23	25.54	9.72	( $G,C_2,3$ )	25	21	22	53	53	53.15	1.60	( $P,G,2$ )	53	54	54
25	24	25.73	8.58	( $G,C_4,2$ )	26	16	24	54	53	53.15	1.67	( $P,G,3$ )	55	55	55
26	26	25.78	8.31	( $G,C_4^{P2P},2$ )	24	19	25	55	54	52.95	2.79	( $C,G,1$ )	54	53	52
27	28	31.21	12.23	( $G,G,2$ )	27	24	29	56	56	55.81	1.60	( $C,G,3$ )	57	56	56
28	28	27.89	8.67	( $G,C_4^{P2P},3$ )	28	26	27	57	56	55.83	1.61	( $C,G,2$ )	56	57	57
29	29	24.99	8.25	( $C,C_2^{P2P},1$ )	30	32	31								

Table 4.5: Configuration rankings under WC/L.



Rankings Under Worst-Case Overheads, Idle															
Rank	Median	Avg	$\sigma$	(CPU,GPU, $\rho$ )	AC/I	AC/L	WC/L	Rank	Median	Avg	$\sigma$	(CPU,GPU, $\rho$ )	AC/I	AC/L	WC/L
1	1	3.26	4.68	$(C, P, \infty)$	2	1	1	30	30	27.17	6.89	$(C, C_2^{P2P}, 1)$	32	31	29
2	3	4.84	4.68	$(C, P, 2)$	5	4	2	31	30	27.58	9.47	$(G, G, 3)$	27	30	33
3	4	5.12	5.02	$(C, P, 3)$	3	3	3	32	32	29.47	7.17	$(C, C_2, 1)$	34	32	31
4	5	8.52	7.28	$(G, P, \infty)$	1	2	4	33	32	31.23	5.60	$(G, C_2^{P2P}, 1)$	31	33	32
5	7	11.05	8.67	$(G, P, 2)$	6	6	7	34	34	32.88	6.26	$(G, C_2, 1)$	33	34	35
6	7	11.19	8.91	$(G, P, 3)$	4	5	6	35	35	31.47	10.13	$(C, C_4^{P2P}, 1)$	36	35	34
7	9	12.68	8.43	$(C, C_2^{P2P}, 2)$	10	8	8	36	36	34.31	8.63	$(G, C_4^{P2P}, 1)$	35	36	39
8	10	12.29	9.67	$(P, P, \infty)$	7	7	5	37	37	32.94	11.03	$(C, C_4, 1)$	38	37	36
9	11	14.38	8.11	$(C, C_2, 2)$	11	9	9	38	38	35.61	9.63	$(G, C_4, 1)$	37	38	40
10	12	14.28	10.24	$(P, P, 2)$	13	16	11	39	39	37.59	5.82	$(P, C_2^{P2P}, 1)$	40	39	37
11	13	14.20	6.52	$(C, P, 1)$	18	10	12	40	40	36.72	13.48	$(G, G, 1)$	39	40	45
12	13	15.14	9.74	$(P, P, 3)$	14	14	10	41	40	38.40	5.18	$(P, C_2, 1)$	41	41	38
13	15	15.51	6.79	$(C, C_4^{P2P}, 2)$	15	15	16	42	41	39.64	5.05	$(P, C_2^{P2P}, 2)$	42	43	42
14	15	15.62	8.76	$(C, C_2^{P2P}, 3)$	20	17	13	43	41	40.10	4.89	$(P, C_2, 2)$	43	42	41
15	15	16.10	6.74	$(C, C_4, 2)$	12	13	15	44	42	41.08	5.02	$(P, C_2^{P2P}, 3)$	44	45	44
16	17	18.79	9.02	$(G, C_2^{P2P}, 2)$	8	11	19	45	43	41.71	4.91	$(P, C_2, 3)$	45	44	43
17	18	17.82	8.01	$(C, C_2, 3)$	23	18	14	46	47	46.48	3.44	$(P, C_4^{P2P}, 1)$	46	46	46
18	19	20.36	8.85	$(G, C_2, 2)$	9	12	20	47	47	47.07	2.76	$(P, C_4, 1)$	47	47	47
19	20	18.82	10.42	$(P, P, 1)$	30	26	21	48	48	48.24	2.12	$(P, C_4^{P2P}, 2)$	49	49	49
20	20	19.17	8.26	$(C, C_4^{P2P}, 3)$	29	23	17	49	48	48.42	1.75	$(P, C_4, 2)$	48	48	48
21	21	20.22	7.55	$(G, P, 1)$	22	19	22	50	49	49.01	1.92	$(P, C_4, 3)$	50	50	50
22	22	20.71	7.79	$(C, C_4, 3)$	28	20	18	51	49	49.10	1.97	$(P, C_4^{P2P}, 3)$	51	51	51
23	22	22.12	8.46	$(G, C_2^{P2P}, 3)$	17	21	23	52	53	52.77	1.95	$(P, G, 1)$	52	53	52
24	22	22.30	6.62	$(G, C_4^{P2P}, 2)$	19	25	26	53	53	53.37	1.15	$(P, G, 2)$	54	54	53
25	23	23.97	8.08	$(G, C_2, 3)$	21	22	24	54	54	52.88	2.84	$(C, G, 1)$	53	52	55
26	24	23.36	6.66	$(G, C_4, 2)$	16	24	25	55	54	53.47	1.31	$(P, G, 3)$	55	55	54
27	25	25.07	8.13	$(G, G, 2)$	24	29	27	56	56	56.29	0.80	$(C, G, 2)$	57	57	57
28	27	24.67	7.49	$(G, C_4^{P2P}, 3)$	26	27	28	57	56	56.29	0.84	$(C, G, 3)$	56	56	56
29	28	26.37	7.50	$(G, C_4, 3)$	25	28	30								

Table 4.6: Configuration rankings under WC/I.

Rankings Under Average-Case Overheads, Loaded															
Rank	Median	Avg	$\sigma$	(CPU,GPU, $\rho$ )	WC/I	AC/I	WC/L	Rank	Median	Avg	$\sigma$	(CPU,GPU, $\rho$ )	WC/I	AC/I	WC/L
1	1	2.07	3.08	(C, P, $\infty$ )	1	2	1	30	30	29.02	10.89	(G, G, 3)	31	27	33
2	2	3.91	5.58	(G, P, $\infty$ )	4	1	4	31	30	26.95	7.65	(C, C <sub>2</sub> <sup>P2P</sup> , 1)	30	32	29
3	4	5.47	4.74	(C, P, 3)	3	3	3	32	32	28.98	8.17	(C, C <sub>2</sub> , 1)	32	34	31
4	4	5.80	4.42	(C, P, 2)	2	5	2	33	32	30.49	6.66	(G, C <sub>2</sub> <sup>P2P</sup> , 1)	33	31	32
5	5	9.20	9.96	(G, P, 3)	6	4	6	34	34	32.50	6.95	(G, C <sub>2</sub> , 1)	34	33	35
6	6	9.80	9.77	(G, P, 2)	5	6	7	35	35	29.86	10.93	(C, C <sub>4</sub> <sup>P2P</sup> , 1)	35	36	34
7	9	11.39	8.59	(P, P, $\infty$ )	8	7	5	36	35	33.08	9.28	(G, C <sub>4</sub> <sup>P2P</sup> , 1)	36	35	39
8	11	13.78	7.57	(C, C <sub>2</sub> <sup>P2P</sup> , 2)	7	10	8	37	37	31.29	11.81	(C, C <sub>4</sub> , 1)	37	38	36
9	11	13.95	7.51	(C, C <sub>2</sub> , 2)	9	11	9	38	37	34.69	9.80	(G, C <sub>4</sub> , 1)	38	37	40
10	15	16.32	6.87	(C, P, 1)	11	18	12	39	39	38.02	6.12	(P, C <sub>2</sub> <sup>P2P</sup> , 1)	39	40	37
11	15	18.15	9.80	(G, C <sub>2</sub> <sup>P2P</sup> , 2)	16	8	19	40	39	38.50	11.84	(G, G, 1)	40	39	45
12	15	18.54	10.05	(G, C <sub>2</sub> , 2)	18	9	20	41	40	38.76	5.75	(P, C <sub>2</sub> , 1)	41	41	38
13	16	16.66	6.44	(C, C <sub>4</sub> , 2)	15	12	15	42	41	39.46	5.78	(P, C <sub>2</sub> , 2)	43	43	41
14	16	17.14	8.12	(P, P, 3)	12	14	10	43	41	39.63	5.94	(P, C <sub>2</sub> <sup>P2P</sup> , 2)	42	42	42
15	16	17.46	7.01	(C, C <sub>4</sub> <sup>P2P</sup> , 2)	13	15	16	44	42	40.51	5.80	(P, C <sub>2</sub> , 3)	45	45	43
16	16	17.71	8.80	(P, P, 2)	10	13	11	45	42	40.77	5.81	(P, C <sub>2</sub> <sup>P2P</sup> , 3)	44	44	44
17	17	16.53	7.45	(C, C <sub>2</sub> <sup>P2P</sup> , 3)	14	20	13	46	47	45.99	4.67	(P, C <sub>4</sub> <sup>P2P</sup> , 1)	46	46	46
18	18	17.29	7.30	(C, C <sub>2</sub> , 3)	17	23	14	47	47	46.46	4.17	(P, C <sub>4</sub> , 1)	47	47	47
19	20	20.19	8.61	(G, P, 1)	21	22	22	48	48	47.22	4.10	(P, C <sub>4</sub> , 2)	49	48	48
20	21	20.36	7.37	(C, C <sub>4</sub> , 3)	22	28	18	49	48	47.34	4.47	(P, C <sub>4</sub> <sup>P2P</sup> , 2)	48	49	49
21	21	21.32	9.89	(G, C <sub>2</sub> <sup>P2P</sup> , 3)	23	17	23	50	49	47.68	4.19	(P, C <sub>4</sub> , 3)	50	50	50
22	21	21.64	9.89	(G, C <sub>2</sub> , 3)	25	21	24	51	49	47.72	4.73	(P, C <sub>4</sub> <sup>P2P</sup> , 3)	51	51	51
23	21	20.15	7.32	(C, C <sub>4</sub> <sup>P2P</sup> , 3)	20	29	17	52	53	52.81	2.92	(C, G, 1)	54	53	55
24	22	21.99	8.01	(G, C <sub>4</sub> , 2)	26	16	25	53	53	53.01	2.17	(P, G, 1)	52	52	52
25	22	22.31	7.67	(G, C <sub>4</sub> <sup>P2P</sup> , 2)	24	19	26	54	53	53.26	2.05	(P, G, 2)	53	54	53
26	25	23.86	8.39	(P, P, 1)	19	30	21	55	54	53.32	1.97	(P, G, 3)	55	55	54
27	26	24.14	8.43	(G, C <sub>4</sub> <sup>P2P</sup> , 3)	28	26	28	56	56	55.86	1.84	(C, G, 3)	57	56	56
28	26	24.25	8.60	(G, C <sub>4</sub> , 3)	29	25	30	57	56	55.89	1.80	(C, G, 2)	56	57	57
29	27	27.32	10.60	(G, G, 2)	27	24	27								

Table 4.7: Configuration rankings under AC/L.

Rankings Under Average-Case Overheads, Idle															
Rank	Median	Avg	$\sigma$	(CPU,GPU, $\rho$ )	WC/I	AC/L	WC/L	Rank	Median	Avg	$\sigma$	(CPU,GPU, $\rho$ )	WC/I	AC/L	WC/L
1	2	3.12	2.49	$(G,P,\infty)$	4	2	4	30	25	23.74	7.54	$(P,P,1)$	19	26	21
2	2	3.71	4.83	$(C,P,\infty)$	1	1	1	31	31	29.31	7.79	$(G,C_2^{P2P},1)$	33	33	32
3	4	5.90	5.54	$(C,P,3)$	3	3	3	32	32	30.84	4.16	$(C,C_2^{P2P},1)$	30	31	29
4	5	5.08	3.37	$(G,P,3)$	6	5	6	33	33	30.45	8.10	$(G,C_2,1)$	34	34	35
5	5	6.21	5.39	$(C,P,2)$	2	4	2	34	33	31.86	4.63	$(C,C_2,1)$	32	32	31
6	6	5.74	3.09	$(G,P,2)$	5	6	7	35	36	32.20	10.74	$(G,C_4^{P2P},1)$	36	36	39
7	10	13.44	9.30	$(P,P,\infty)$	8	7	5	36	36	33.89	7.44	$(C,C_4^{P2P},1)$	35	35	34
8	12	15.23	8.27	$(G,C_2^{P2P},2)$	16	11	19	37	37	33.26	11.35	$(G,C_4,1)$	38	38	40
9	12	15.46	8.49	$(G,C_2,2)$	18	12	20	38	37	34.84	7.86	$(C,C_4,1)$	37	37	36
10	12	15.60	8.50	$(C,C_2^{P2P},2)$	7	8	8	39	39	34.13	15.47	$(G,G,1)$	40	40	45
11	12	15.99	8.73	$(C,C_2,2)$	9	9	9	40	39	38.95	3.83	$(P,C_2^{P2P},1)$	39	39	37
12	16	16.80	7.02	$(C,C_4,2)$	15	13	15	41	40	39.44	3.40	$(P,C_2,1)$	41	41	38
13	16	16.88	9.51	$(P,P,2)$	10	16	11	42	42	40.90	3.17	$(P,C_2^{P2P},2)$	42	43	42
14	17	17.69	8.74	$(P,P,3)$	12	14	10	43	42	40.99	3.04	$(P,C_2,2)$	43	42	41
15	17	18.04	7.24	$(C,C_4^{P2P},2)$	13	15	16	44	43	42.18	3.01	$(P,C_2^{P2P},3)$	44	45	44
16	18	18.29	6.59	$(G,C_4,2)$	26	24	25	45	43	42.21	3.15	$(P,C_2,3)$	45	44	43
17	18	18.69	8.86	$(G,C_2^{P2P},3)$	23	21	23	46	47	46.35	3.31	$(P,C_4^{P2P},1)$	46	46	46
18	19	18.84	6.70	$(C,P,1)$	11	10	12	47	47	46.75	3.00	$(P,C_4,1)$	47	47	47
19	19	19.03	7.17	$(G,C_4^{P2P},2)$	24	25	26	48	48	47.85	2.69	$(P,C_4,2)$	49	48	48
20	19	19.03	8.64	$(C,C_2^{P2P},3)$	14	17	13	49	48	47.95	2.79	$(P,C_4^{P2P},2)$	48	49	49
21	19	19.44	8.46	$(G,C_2,3)$	25	22	24	50	49	48.73	2.88	$(P,C_4,3)$	50	50	50
22	20	17.66	8.02	$(G,P,1)$	21	19	22	51	49	48.90	2.90	$(P,C_4^{P2P},3)$	51	51	51
23	20	19.83	8.21	$(C,C_2,3)$	17	18	14	52	53	52.29	2.88	$(P,G,1)$	52	53	52
24	21	19.27	7.98	$(G,G,2)$	27	29	27	53	53	52.33	3.39	$(C,G,1)$	54	52	55
25	24	21.86	7.95	$(G,C_4,3)$	29	28	30	54	53	52.97	2.52	$(P,G,2)$	53	54	53
26	24	21.97	7.95	$(G,C_4^{P2P},3)$	28	27	28	55	54	53.13	2.52	$(P,G,3)$	55	55	54
27	25	22.66	9.34	$(G,G,3)$	31	30	33	56	56	55.81	2.26	$(C,G,3)$	57	56	56
28	25	23.03	6.53	$(C,C_4,3)$	22	20	18	57	56	55.83	2.24	$(C,G,2)$	56	57	57
29	25	23.35	6.77	$(C,C_4^{P2P},3)$	20	23	17								

Table 4.8: Configuration rankings under AC/L.

would both have a value of “1.” They do not. This result demonstrates that configurations other than  $(C, P, \infty)$  perform better for some classes of task sets.

**Observation 15.** *Under partitioned GPUs, schedulability tended to be maximized when  $\rho$  was large, especially when  $\rho = \infty$ .*

We may observe this by scanning the system configuration columns in Tables 4.5, 4.6, 4.7, and 4.8, picking out entries matching  $(*, P, *)$ . Observe that entries that only differ by  $\rho$  generally tend to be ranked in decreasing  $\rho$ -order. For instance, in Table 4.7, the configurations  $(C, P, \infty)$ ,  $(C, P, 3)$ ,  $(C, P, 2)$ , and  $(C, P, 1)$  are ranked first, third, fourth, and tenth, respectively. There are minor exceptions to this general trend. For example, in Table 4.5,  $(C, P, 2)$  is ranked 2<sup>nd</sup>, while  $(C, P, 3)$  is ranked 3<sup>rd</sup>. However, the average sub-ranks of these configurations are very close: 5.72 and 5.73, respectively. We see similar exceptions for  $(*, P, *)$  configurations, where those with  $\rho = 2$  are occasionally ranked slightly higher than the similar corresponding configuration with  $\rho = 3$ . This occurs under all overhead assumptions, so we cannot conclude that these exceptions are due to overheads alone. Despite these exceptions, the general trend still holds: under partitioned GPU scheduling, configurations with many GPU tokens perform best. This is a good property, since it motivates the use of GPUSync configurations that maximize the opportunity for parallelism at runtime.

**Observation 16.** *With the exception of those where CPUs are partitioned, schedulability of clustered GPU configurations tended to be maximized when  $\rho = 2$ .*

To see this, locate the sets of clustered GPU configurations that only differ by their values for  $\rho$  in Tables 4.5, 4.6, 4.7, and 4.8. With the exception of those with partitioned CPUs, entries where  $\rho = 2$  have the highest rank among similar configuration that only differ by  $\rho$ . For example, in Table 4.5,  $(C, C_4, 2)$  is ranked 15<sup>th</sup>, while  $(C, C_4, 3)$  is ranked 18<sup>th</sup>, and  $(C, C_4, 1)$  is ranked 36<sup>th</sup>. Similar trends can be observed for rankings in the WC/L, AC/I, and AC/L columns, as well. This result is interesting because it indicates, in terms of schedulability, that there is a “sweet spot” to the number of tokens for many clustered GPU configurations that maximizes theoretical performance. As we see later in Chapter 5, the existence of token-number sweet spots can be observed in runtime performance as well (see Observation 45).

**Observation 17.** *Peer-to-peer migrations offered better schedulability than system memory migrations.*

With the exception of configurations that use a GPU Allocator based upon the CK-OMLP, every clustered GPU configuration where peer-to-peer migrations are used ranks higher than the similar configuration that use system memory migration. In most cases, the system-memory-variant ranks closely below the corresponding peer-to-peer configuration. For instance,  $(G, C_2^{P2P}, 2)$  ranks 16<sup>th</sup> while  $(G, C_2, 2)$  ranks 18<sup>th</sup> in Table 4.6. The differences in ranking between similarly matched configurations under the other overhead data sets can also be observed in Tables 4.5, 4.7, and 4.8.

This result differs from what we reported in prior work (Elliott *et al.*, 2013). In that work, results from schedulability experiments showed that *system memory migration* configurations outperformed corresponding peer-to-peer configurations. The schedulability analysis for peer-to-peer configurations used in those experiments was based upon the non-ILP fine-grain blocking analysis we described in Section 4.3.4.3. In contrast, the schedulability analysis we use in *this* work employs our ILP-based blocking chain analysis. The improvement in our results demonstrates the benefits of our ILP-based analysis. Also, as we see later in Section 4.4, these improvements bring our analytical results more in line with observed runtime behavior.

**Observation 18.** *Smaller GPU clusters, where  $g = 2$ , offered better schedulability than larger clusters, where  $g = 4$ .*

This observation holds for every evaluated configuration, even those that use a GPU Allocator based upon the CK-OMLP, which have bucked many other trends observed here. We give a diverse set of examples. Under WC/L overheads in Table 4.5,  $(P, C_2^{P2P}, 3)$  (ranked 44<sup>th</sup>) has an average sub-ranking of 36.35, while  $(P, C_4^{P2P}, 3)$  (ranked 51<sup>st</sup>) has an average sub-ranking of 47.14. Under WC/I overheads in Table 4.6,  $(C, C_2, 2)$  (ranked 9<sup>th</sup>) has an average sub-ranking of 14.38, while  $(C, C_4, 2)$  (ranked 15<sup>th</sup>) has an average sub-ranking of 16.10. Under AC/L overheads in Table 4.7,  $(G, C_2^{P2P}, 2)$  (ranked 11<sup>th</sup>) has an average sub-ranking of 18.15, while  $(G, C_4^{P2P}, 2)$  (ranked 25<sup>th</sup>) has an average sub-ranking of 22.31. Finally, under AC/I overheads in Table 4.8,  $(C, C_2, 1)$  (ranked 34<sup>th</sup>) has an average sub-ranking of 31.86, while  $(C, C_4, 1)$  (ranked 38<sup>th</sup>) has an average sub-ranking of 34.84.

There are two aspects of larger GPU clusters that explain this poor performance. First, for larger clusters that use peer-to-peer migration, in Section 4.3.1.2 (see Observation 8), DMA overhead costs when  $g = 4$  are approximately twice those when  $g = 2$ . Second, with larger GPU clusters under *both* peer-to-peer and system memory migration configurations, jobs that issue token and engine requests may experience worse blocking in the worst-case. This is not only explained by the asymptotic blocking bounds for token and engine requests

that increase with  $g$ , but also by the fact that a single request potentially competes with the *requests of more tasks* due to there being fewer GPU clusters. Since blocking analysis always assumes worst-case resource request scenarios, the poor performance of larger cluster sizes is usually inevitable.

**Observation 19.** *The clustered CPU and clustered GPU configuration,  $(C, C^{P2P}, 2)$ , was competitive with partitioned CPU and partitioned GPU configurations,  $(P, P, *)$ .*

Under WC/L overheads, in Table 4.5 we see that  $(P, P, \infty)$  is ranked 5<sup>th</sup>, while  $(C, C^{P2P}, 2)$  is ranked 8<sup>th</sup>. Under AC/L overheads, in Table 4.7, this rankings gap shrinks:  $(P, P, \infty)$  is ranked 7<sup>th</sup>, while  $(C, C^{P2P}, 2)$  remains ranked 8<sup>th</sup>. The relative performance of these configurations switch under WC/I overheads, as we see in Table 4.6. Here,  $(C, C^{P2P}, 2)$  is ranked 7<sup>th</sup>, and  $(P, P, \infty)$  is ranked 8<sup>th</sup>. Under AC/I overheads, in Table 4.8 we see that  $(P, P, \infty)$  is ranked 7<sup>th</sup> once again, while  $(C, C^{P2P}, 2)$  is ranked 10<sup>th</sup>.

We have already established that configurations  $(C, P, \infty)$  and  $(G, P, \infty)$  have the best schedulability in Observation 15. Why do we care about the comparative performance of  $(C, C^{P2P}, 2)$  against fully partitioned alternatives? As we see later in Section 4.4.2, GPUSync’s affinity-aware heuristics greatly reduce the likelihood of GPU migrations. As a result, GPUSync configurations with clustered GPUs can outperform partitioned alternatives at runtime on average. We compare the best-performing clustered configuration,  $(C, C^{P2P}, 2)$ , against fully partitioned alternatives because these partitioned configurations represent the approach one would expect from current industrial practice—one where all computations are statically assigned to processor partitions. We highlight the analytical performance of  $(C, C^{P2P}, 2)$  to show that a system designer need only sacrifice a small degree of schedulability to realize potential runtime benefits of clustered GPUs, with respect to the *de facto* alternative.

**Observation 20.** *Schedulability was comparably poor under GPU Allocators based upon the CK-OMLP.*

We observe in Table 4.5 that configurations  $(P, G, *)$ ,  $(P, C, *)$ ,  $(P, C^{P2P}, *)$ , and  $(C, G, *)$  make up 18 of the 21 *lowest* ranked configurations under WC/L. Similar trends hold under the other overhead data sets, as can be seen in Tables 4.6, 4.7, and 4.8. The consistently poor performance of configurations that rely upon the CK-OMLP for GPU token assignment clearly indicates that these configurations should be avoided (at least in the absence of other compelling system requirements).

Recall from Section 2.1.7.3 what we know of the CK-OMLP: tasks that do not obtain a resource through the CK-OMLP may still suffer s-oblivious pi-blocking due to priority donation. In general, the utilization

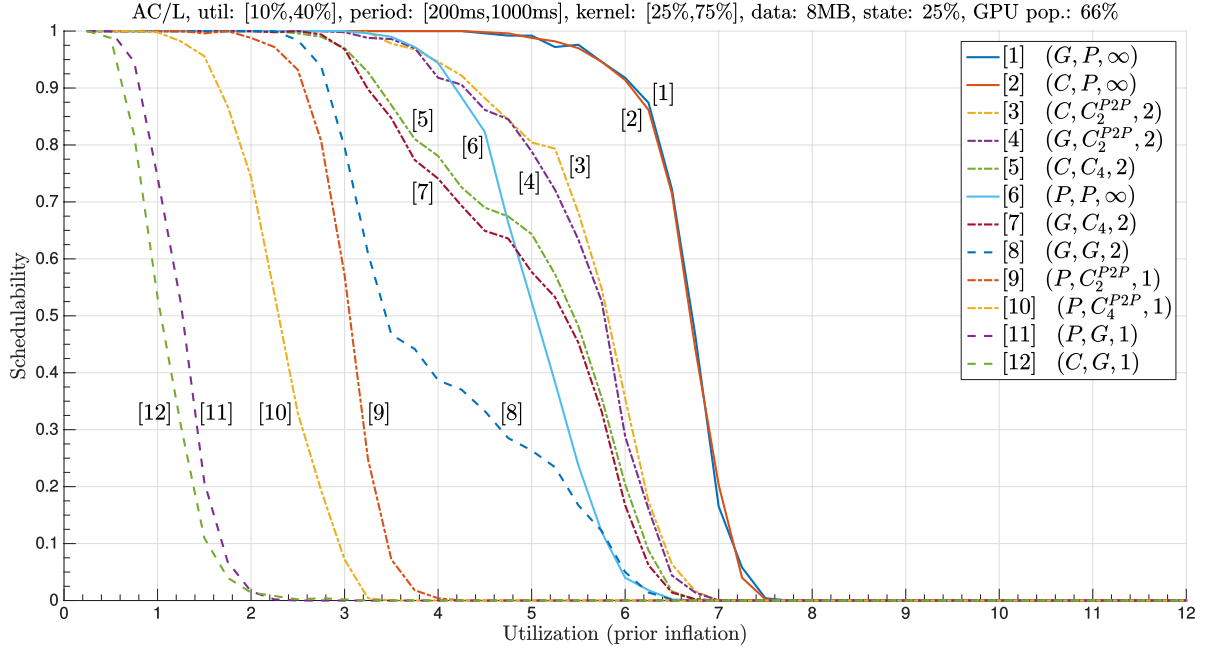


Figure 4.16: Detailed schedulability result.

of every task that may share a CPU with a GPU-using tasks increases under the CK-OMLP—this severely decreases schedulability. This explains the poor schedulability we see here.

This completes our high-level comparisons of the various system configurations. We now take a deeper look at some of our results.

Figure 4.16 plots schedulability curves for the twelve highest-ranked configuration of each high-level GPUSync configuration under AC/L overheads. The curves are numbered in descending order according to the area under each curve. That is, curve 1 has the greatest area under its curve, while curve 12 has the least area under its curve. In general, a curve with greater area reflects better performance in terms of schedulability. The tasks of the schedulability experiment illustrated by Figure 4.16 had *medium* utilizations, *moderate* periods, *heavy* data requirements, and each GPU-using task had a state size of 2MB (or 25% of the tasks’ data requirement). GPU kernel execution times were sampled from a [25%,75%] uniform distribution. Finally, 66% of the tasks in each task set used a GPU. Figure 4.16 represents a tiny portion of our schedulability results. However, we examine these schedulability curves to reinforce the observations we made from the ranking data and illustrate additional points. We make the following observations.

**Observation 21.** *The clustered and global CPU configurations with partitioned GPUs gave the best schedulability results.*

Curves 1 and 2 plot schedulability for the configurations  $(G, P, \infty)$  and  $(C, P, \infty)$ , respectively. As we see in Figure 4.16, these curves nearly completely overlap, reflecting roughly equivalent performance. We also see in this figure that these curves reflect greater schedulability than those for the other configurations. Although this result is immediately apparent in Figure 4.16, we consider task sets with a utilization of about 5.0 ( $x = 5.0$ ) to make more precise comparisons.

Roughly 98% of task sets with a utilization of about 5.0 are schedulable under  $(G, P, \infty)$  and  $(C, P, \infty)$ . In comparison, only about 80% of task sets with that utilization are schedulable under  $(C, C_2^{P2P}, 2)$  (curve 3) and  $(G, C_2^{P2P}, 2)$  (curve 4). Approximately 65% and 57% of task sets with a utilization of 5.0 are schedulable under  $(C, C_4, 2)$  (curve 5) and  $(G, C_4, 2)$  (curve 7), respectively; and roughly 50% and 25% of task sets with a 5.0 utilization are schedulable under  $(P, P, \infty)$  and  $(G, G, 2)$ , respectively. No task set with a utilization of 5.0 was schedulable under  $(P, C_2^{P2P}, 1)$  (curve 9),  $(P, C_4^{P2P}, 1)$  (curve 10),  $(P, G, 1)$  (curve 11), or  $(C, G, 1)$  (curve 12) configurations.

**Observation 22.** *The conventional configuration,  $(P, P, \infty)$ , did not give the best results.*

We see in Figure 4.16 that,  $(C, C_2^{P2P}, 2)$  (curve 3),  $(G, C_2^{P2P}, 2)$  (curve 4), and  $(P, P, \infty)$  (curve 6) offer similar performance for task sets with utilizations no greater than 4.2. However, these curves begin to diverge thereafter. Ultimately, configurations  $(C, C_2^{P2P}, 2)$  and  $(G, C_2^{P2P}, 2)$  have better schedulability. The configurations  $(C, C_4, 2)$  (curve 5) and  $(G, C_4, 2)$  (curve 7) are arguably competitive with  $(P, P, \infty)$  as well. We see that these two configurations have better schedulability than  $(P, P, \infty)$  for task sets with utilizations greater than approximately 4.8.

The curves in Figure 4.16 provide a concrete example in support of Observation 19, which noted that better alternatives exist to the default industry approach of configuration  $(P, P, \infty)$ .

**Observation 23.** *Small GPU peer-to-peer clusters where  $g = 2$  perform relatively well.*

We make this observation by comparing the curves in Figure 4.16 for similar clustered GPU configurations that differ in cluster size. For example, configuration  $(C, C_2^{P2P}, 2)$  (curve 3) dominates  $(C, C_4, 2)$  (curve 5). Similarly,  $(G, C_2^{P2P}, 2)$  (curve 4) dominates  $(G, C_4, 2)$  (curve 7). Even  $(P, C_2^{P2P}, 1)$  (curve 9) dominates  $(P, C_4^{P2P}, 1)$  (curve 10). This observation reinforces Observation 18: smaller GPU cluster are better than large GPU clusters.

**Observation 24.** *Peer-to-peer GPU clusters do not always offer the best schedulability.*



In Observation 17, we remarked on compelling evidence that GPU clusters with peer-to-peer migration offered better schedulability than similar GPU clusters that use system memory migration. While this remains true in general, as we see in Figure 4.16, there are exceptional cases.

Recall that Figure 4.16 plots only the schedulability curves of the twelve best high-level configurations. We see that configurations  $(C, C_4, 2)$  and  $(G, C_4, 2)$  are plotted by curves 5 and 7, respectively. There are no curves for configuration  $(C, C_4^{P2P}, 2)$  or  $(G, C_4^{P2P}, 2)$ . This is because configurations  $(C, C_4, 2)$  and  $(G, C_4, 2)$  had better schedulability. In this case, the benefits of ILP-based blocking chain analysis do not overcome larger DMA overheads or asymptotically more complex blocking bounds that are associated with large GPU clusters with peer-to-peer migration.

**Observation 25.** *Configurations where GPUs are shared among CPU clusters had inferior schedulability results.*

In Observation 20, we remarked that configurations where the GPU Allocator was based upon the CK-OMLP offered poor schedulability. This claim is supported here by the four lowest curves in Figure 4.16. Configurations  $(P, C_2^{P2P}, 1)$  (curve 9),  $(P, C_4^{P2P}, 1)$  (curve 10),  $(P, G, 1)$  (curve 11), and  $(C, G, 1)$  (curve 12) were unable to schedule task sets that other configurations were always able to schedule. For example, 100% of the evaluated task sets with a utilization of 4.0 were schedulable by configurations  $(G, P, \infty)$  (curve 1) and  $(C, P, \infty)$  (curve 2). In contrast, practically no task sets with a utilization of 4.0 were schedulable under  $(P, C_2^{P2P}, 1)$  (curve 9).

If we ended our study of schedulability results here, one may be left with the impression that GPUs only harm schedulability. In Figure 4.16, no task set with a utilization of 8.0 or more is schedulable, and yet in Section 4.3.2, we stated that task sets that fully utilize platform CPUs are schedulable with bounded deadline tardiness under FL scheduling. Does this mean that no GPUSync configuration supports a computing capacity of eight CPUs on our evaluation platform? The answer to this question is “no.” This is because the  $x$ -axis of Figure 4.16 reflects only *CPU utilization*—it does not reflect the gains in computational capacity made possible by GPUs. We illustrate these gains by considering what we call the *effective utilization* of a task set with GPU-using tasks.

To find the effective utilization of a task set, we begin by supposing a GPU-to-CPU speed-up ratio. Let us denote this ratio by  $\mathcal{S}$ . We then analytically convert each GPU-using task into a functionally equivalent CPU-only independent task by viewing each unit of time spent executing on an EE as  $\mathcal{S}$  time units spent

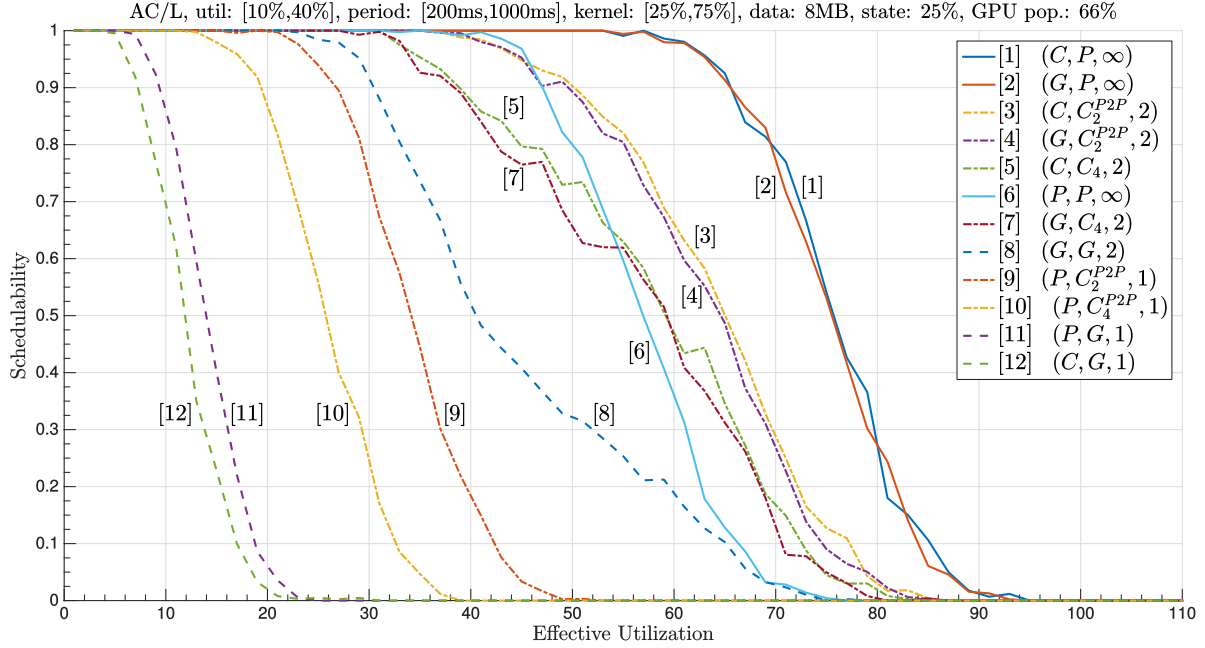


Figure 4.17: A detailed schedulability result showing schedulability and effective utilization.

executing on a CPU. We also discard all DMA operations and ignore all GPU-related overheads. The effective utilization of a task set is the sum of all task utilizations after this conversion process.<sup>17</sup>

We re-plot the schedulability curves of the task sets evaluated in Figure 4.16 in terms of effective utilization in Figure 4.17, where we assume a speed-up ratio of  $\mathcal{S} = 32$ . This is a reasonable speed-up factor, since we are converting the execution time of data-parallel GPU computations to serialized CPU computations. We make the following critical observation.

**Observation 26.** *GPUs greatly increase the computational capacity of a platform.*

Every GPUSync configuration represented by the curves in Figure 4.17 was able to schedule a task set that has an effective utilization greater than 12.0 (the number of system CPUs). The best performing configurations,  $(C, P, \infty)$  (curve 1) and  $(G, P, \infty)$  (curve 2), managed to schedule task sets that had effective utilizations as great as 94.0. In Figure 4.16, we saw that these configurations were not able to schedule task sets with utilizations greater than roughly 8.0—a utilization loss of 4.0. However, this utilization loss is not truly a loss in light of utilization gains from the GPUs. In a way, we sacrifice 4.0 units of CPU capacity in order to gain an equivalent of 90.0 ( $94.0 - 4.0$ ) CPUs from the eight GPUs.

<sup>17</sup>Observe that two task sets with the same real CPU utilization and the same speed-up ratio may still have different effective utilizations, since the tasks in each task set may use the EE to different degrees.

An effective utilization of 94.0 is a best-case schedulability scenario of the configurations represented by Figure 4.17. Even when we consider more conservative cases, the benefits of GPUs remain clear. For instance, about 90% of task sets with effective utilizations of 66.0 were schedulable under  $(C, P, \infty)$  (curve 1) and  $(G, P, \infty)$  (curve 2). Roughly 50% of task sets with effective utilizations of 76.0 were schedulable under  $(C, P, \infty)$  (curve 1) and  $(G, P, \infty)$  (curve 2).

Configurations with clustered GPU scheduling also realized significant gains in computational capacity. About 90% of task sets with effective utilizations of 50.0 were schedulable under  $(C, C_2^{P2P}, 2)$  (curve 3) and  $(G, C_2^{P2P}, 2)$  (curve 4). Roughly 50% of task sets with effective utilizations of 65.0 were schedulable under these configurations.

*This result supports a major part of this dissertation’s thesis: increases in computational capacity outweigh analytical costs introduced by management overheads and limitations of GPU hardware and software.*

This concludes our schedulability analysis of GPUSync.

#### 4.4 Runtime Evaluation

In this section, we evaluate GPUSync through runtime experiments. We present our evaluation in two parts. First, we examine the effectiveness GPUSync’s budgeting mechanisms, Cost Predictor, and affinity-aware GPU Allocator. Second, we assess the effectiveness of clustered GPU management and peer-to-peer migrations through the scheduling of computer vision workloads. Before continuing, we note that we evaluated schedulability under FL schedulers in the prior section, since the analytical techniques associated with FL schedulers provide better bounds on deadline tardiness. However, the use of FL schedulers in practice requires offline analysis (*e.g.*, execution time analysis and derivation of blocking terms) in order to properly determine task priority points. In this section, we forgo such analysis because FL scheduling is not a core aspect of GPUSync. Instead, we evaluate GPUSync under EDF and RM schedulers, since these are easier to use in practice. Up to this point, we have focused mainly on deadline-based scheduling (*e.g.*, FL and EDF). We include RM schedulers in our runtime evaluations, since fixed-priority schedulers are more prevalent in RTOSs than deadline-based schedulers. Moreover, evaluations under RM schedulers also highlight the flexibility of GPUSync’s design. However, as we discuss later in Chapter 6, we leave the

development of overhead-aware schedulability tests for GPUSync under RM schedulers as future work. Such analysis is not necessary in order to perform runtime experiments.

#### 4.4.1 Budgeting, Cost Prediction, and Affinity

We used a mixed task set of CPU-only and GPU-using implicit-deadline periodic tasks to evaluate budget enforcement, the Cost Predictor, and the GPU Allocator. We now describe our task set in more detail.

Numerical code was executed by tasks on both CPUs and GPUs to *simulate* real applications. Task execution time was tightly controlled through the use of processor cycle counters on *both* CPUs and GPUs. GPU-using tasks also transmitted data on the PCIe bus. Task periods ranged from 10ms to 75ms, reflecting a range of periods found in ADAS systems, such as those we described in Chapter 1 (see Table 1.1).

The task set consisted of 28 CPU-only tasks and 34 GPU-using tasks. Each task was assigned a utilization based upon the combined processor time (CPU and GPU engines) a task’s job must receive before completing. Of the CPU-only tasks, twelve had a utilization of 0.1, eight had a utilization of 0.2, and two had a utilization of 0.3. Of the GPU-using tasks, fourteen had a utilization of 0.1, fourteen more had a utilization of 0.25, and six had a utilization of 0.5. 90% of each GPU-using task’s utilization was devoted to GPU operations, with 75% of that towards GPU kernels, and the remaining 25% towards memory copies. Each GPU-using job had one GPU critical section. Within its critical section, each such job executed two GPU kernels of equal execution time. The amount of memory to copy was determined based upon desired copy time and worst-case bus congestion bandwidth derived from empirical measurements. Memory copies were evenly split between input and output data. Task state size was set to twice the combined size of input and output data. Additionally, DMA operations were broken up into 2MB chunks.

We configured our system to run as a cluster along NUMA boundaries. Thus, there were two clusters of six CPUs and four GPUs apiece. The above task set was evenly partitioned between the two clusters. The task set was scheduled under C-EDF and C-RM schedulers. Three tokens ( $\rho = 3$ ) were allocated to each GPU in order to allow all GPU engines to be used simultaneously. Under C-EDF, we configured the GPU Allocator to be optimal under suspension-oblivious analysis, *i.e.*, the GPU Allocator was configured as the R<sup>2</sup>DGLP. We realized this configuration by setting the maximum length of the FIFO queues of the GPU Allocator to  $f = \lceil c/(g \cdot \rho) \rceil = \lceil 6/(4 \cdot 3) \rceil = 1$ . Under C-RM, the GPU Allocator was configured to be strictly priority-ordered, as is common to locking protocols under fixed-priority schedulers. We realized this configuration by setting  $f$  to the minimum size, or  $f = 1$ . These two configurations of the GPU Allocator are actually one in

the same, since our values for  $\rho$  and GPU cluster size are so great. The GPU Allocator under the C-EDF and C-RM schedulers only differ by the method used to determine job priority (earliest-deadline-first for C-EDF and shortest-period-first for C-RM). We used different engine lock configurations for each scheduler. Engine locks were FIFO-ordered and priority-ordered under C-EDF and C-RM schedulers, respectively. GPU migrations were performed using peer-to-peer DMA operations.

The task set was scheduled under two execution-behavior scenarios. Under the first scenario, tasks adhered to their prescribed execution times as closely as possible. Under the second scenario, however, eight GPU-using tasks in each cluster were configured to exhibit *aberrant behaviors* by executing for *ten times* their normal execution time at random moments (roughly spaced by five seconds for each task). These scenarios were scheduled for 180 seconds under both C-EDF and C-RM schedulers and measurements were taken. We use data gathered during the execution of these scenarios as the basis of the evaluations described in Sections 4.4.1.1, 4.4.1.2, and 4.4.1.3.

#### 4.4.1.1 Budget Performance

We analyze the ability of GPUSync to manage budgets (and penalize overrunning tasks) by examining the long-term utilization of the execution engines. Here, we test GPUSync under the early releasing policy described in Section 3.2.4, and compare performance against a no-budget-enforcement policy. We measure execution engine utilization (GPU execution time divided by period) with respect to the *hold time* of execution engine locks. This is an effective measure, even if the engine may idle while the engine lock is held, since all other tasks are blocked from using the engine.

From the task set described in Section 4.4.1, we focus our attention on two of the 34 GPU-using tasks,  $T_1$  and  $T_2$ .  $T_1$  has a period of 15ms and a utilization of 0.25.  $T_1$ 's ideal execution-engine utilization is 0.169, and it sends and receives 512KB to and from the GPU.  $T_2$  has a period of 75ms, a utilization of 0.1, an ideal execution-engine utilization of 0.068, and it sends and receive 1024KB to and from the GPU. We focus on these tasks because of their short and long periods, respectively.

Figure 4.18 depicts the accumulated time (on the y-axis) tasks  $T_1$  and  $T_2$  hold an execution engine lock over the duration of their execution (on the x-axis). Figure 4.18 also displays the equation for the line-of-best-fit of each plotted line. The slope of each line-of-best-fit (*i.e.*, the coefficient of the variable  $x$ ) approximates the accumulated time the associated task holds an engine lock, divided by task period. We interpret this slope as a measure of long-term execution engine utilization. We make three observations.

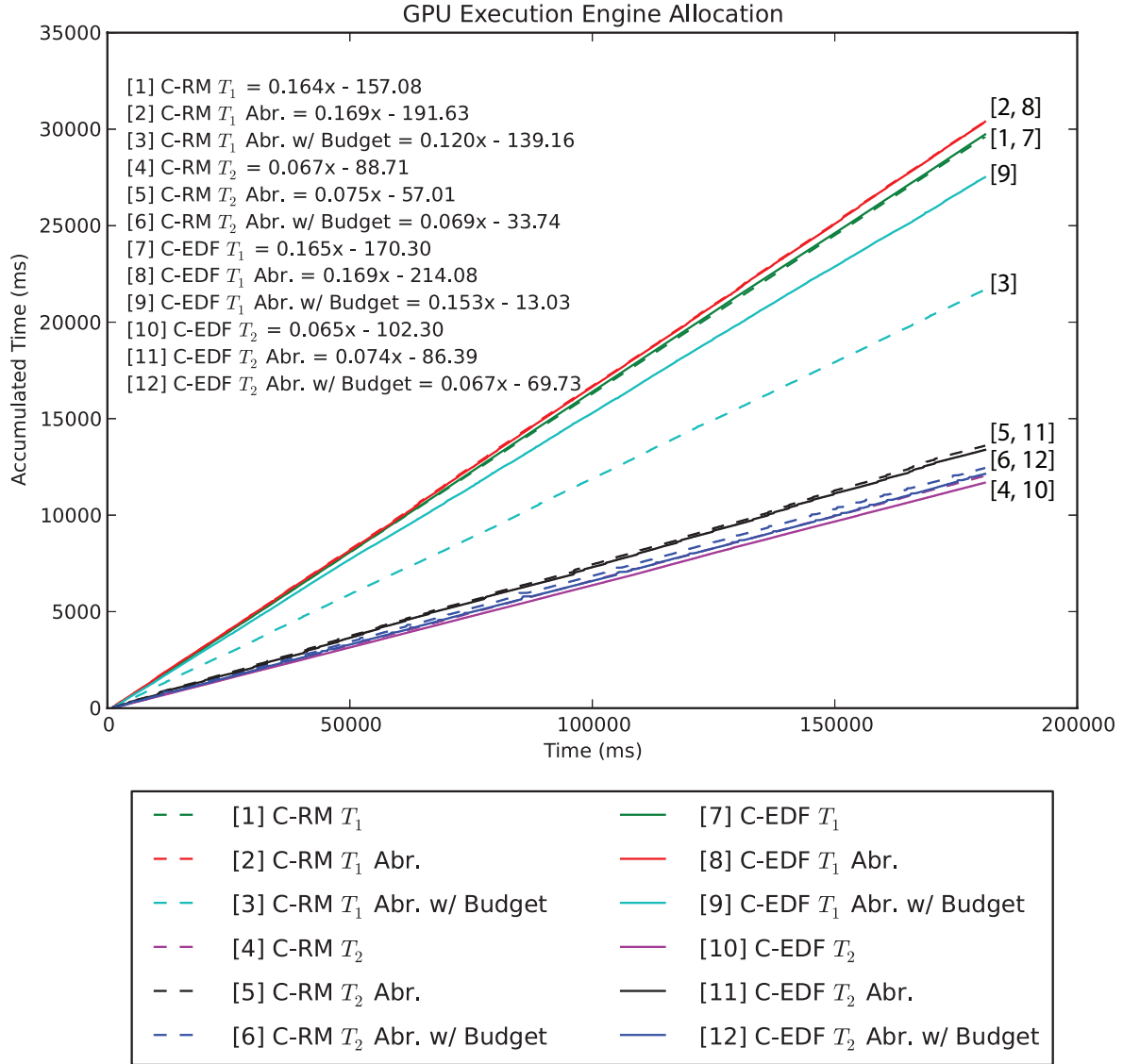


Figure 4.18: Allocated execution engine time.

**Observation 27.** *A synchronization-based approach to GPU scheduling is effective at supplying GPU-using tasks with provisioned execution time.*

Ideally, the slope of the lines in Figure 4.18 should be equal to the task’s execution-engine utilization. With the exception of line 3, the slopes of all the lines are very close to the desired utilization. For example, when  $T_2$  is well-behaved under C-EDF scheduling (line 10), the slope is 0.065, this is commensurate with the assigned utilization of 0.068

**Observation 28.** *Budget enforcement can penalize aberrant tasks by allocating less execution time.*

This may be observed in lines 3 and 9 for the aberrant task  $T_1$  under both C-RM and C-EDF scheduling in Figure 4.18. As shown by line 3,  $T_1$ ’s utilization is 0.12—30% less than the provisioned 0.169, for C-RM. Similarly, for C-EDF,  $T_1$ ’s utilization is 0.153—10% less than the provisioned 0.169, This loss of utilization is a result of the early releasing budget policy where any surplus from an early-released budget is *discarded* after an overrunning job completes.

**Observation 29.** *C-RM and C-EDF can both perform well.*

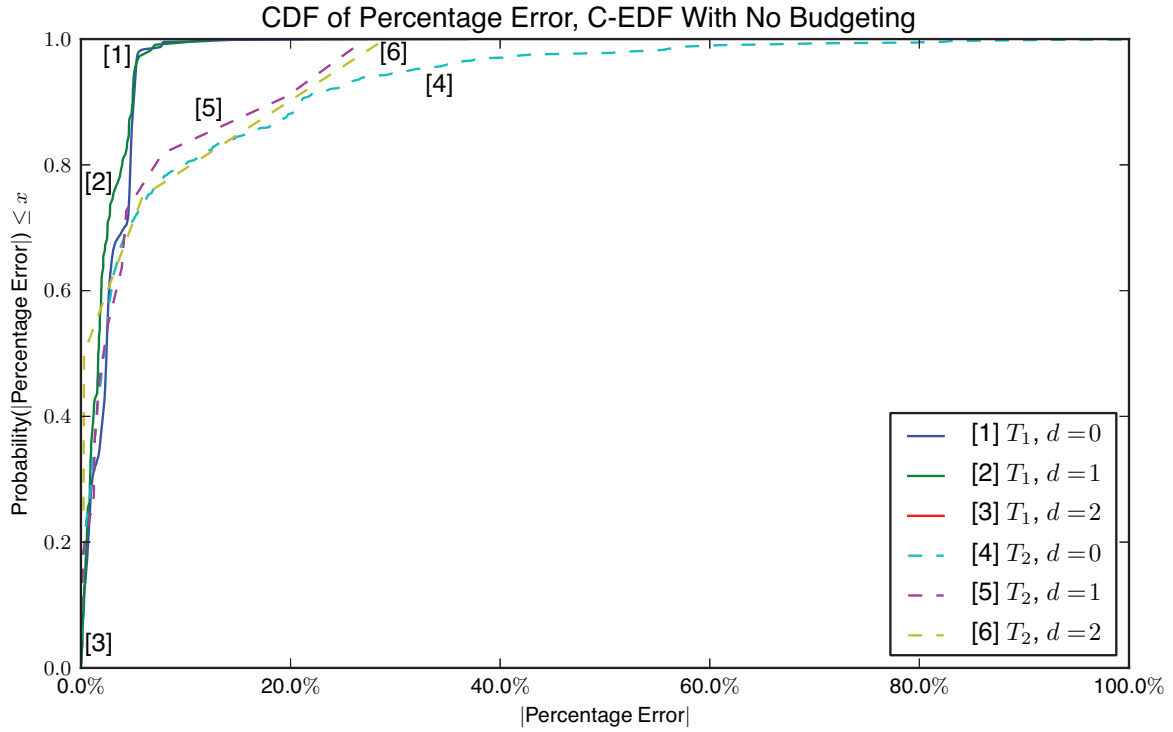
For this particular experiment, we observe that both C-RM and C-EDF are able to supply the needed GPU execution time. This is an important result because it empowers system designers to select the scheduler that suits their needs.

#### 4.4.1.2 Cost Predictor Accuracy

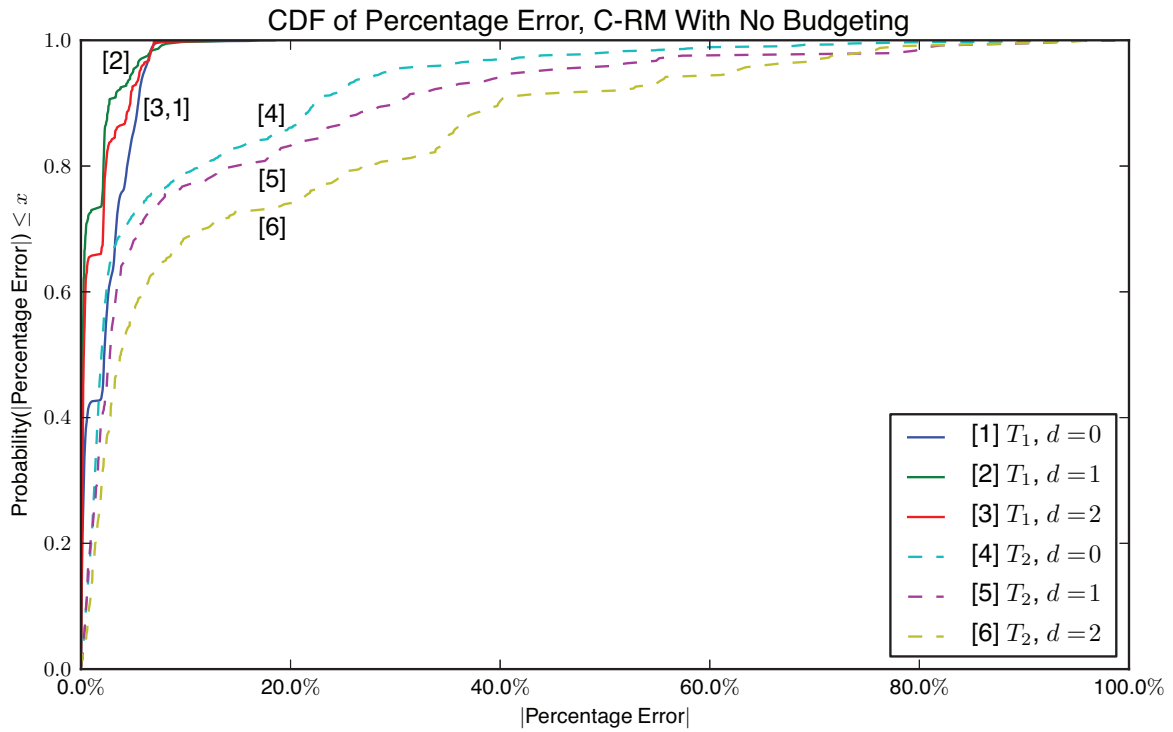
We measured the overheads related to the Cost Predictor because of our concern that computing averages and standard deviations can be computationally expensive. However, we found these worries to be unfounded. Updating the Cost Predictor estimate took  $0.294\mu\text{s}$  on average, and  $2.335\mu$  in the (observed) worst-case.

We now discuss the accuracy of the Cost Predictor. Figure 4.19 plots cumulative distribution functions (CDFs) for the percentage error of the cost predictor for different migration distances under the aberrant behavior scenario, without budget enforcement. Migration distance is denoted by  $d$ ;  $d = 0$  reflects no migration,  $d = 1$  denotes migrations to neighboring (*i.e.*, near) GPUs, and  $d = 2$  reflects migrations to distant (*i.e.*, far) GPUs. We continue to consider tasks  $T_1$  and  $T_2$  described earlier in Section 4.4.1.

**Observation 30.** *The Cost Predictor is generally accurate at predicating token hold time, despite aberrant task behavior.*



(a) C-EDF



(b) C-RM

Figure 4.19: CDFs of percentage error in cost predictions.



As seen in Figure 4.19, under C-EDF (inset(a)), roughly 95% of all predictions for task  $T_1$  have a percentage error of 5% or less (only one sample exists for  $T_1$  where  $d = 2$ ). Accuracy is even better for  $T_1$  under C-RM. This is expected since  $T_1$  has a statically high priority due to its short period, and it is thus able to avoid more interference from other tasks than under C-EDF.

**Observation 31.** *The Cost Predictor is less accurate for tasks with longer execution times and periods.*

Both Figure 4.19(a) and Figure 4.19(b) show that the Cost Predictor is less accurate for  $T_2$  than  $T_1$  in this experiment. This is because  $T_2$  has a longer execution time and period than most other tasks in the task set. Thus,  $T_2$  is more likely to experience interference from aberrant tasks.

**Observation 32.** *The Cost Predictor is moderately less accurate under C-RM than C-EDF.*

This can be seen by comparing insets (a) and (b) of Figure 4.19. For example, about 90% of predictions for  $T_2$  with  $d = 1$  under C-EDF (inset (a)) have a percentage error no greater than 20%. Compare this to C-RM (inset(b)), where only 80% of  $d = 1$  predictions for  $T_2$  have the same degree of accuracy.

**Observation 33.** *The Cost Predictor is generally less accurate for longer migration distances.*

A migrating job of a task must acquire additional copy engine locks and do more work than non-migrating jobs. This introduces variability into token hold times predicted by the Cost Predictor. We see that this generally has a negative effect on the Cost Predictor. This is clearly demonstrated in Figure 4.19(b) for C-RM, where each CDF generally upper-bounds the CDF of the same task at the next migration distance. We note that this does not always hold under C-EDF (Figure 4.19(a)). For example, predictions for  $T_2$  with  $d = 1$  are more accurate than  $d = 0$ . However, as we discuss shortly, this may be due to a smaller sample size of observations.

#### 4.4.1.3 Migration Frequency

The Cost Predictor is an important component of GPUSync, since it influences the behavior of the migration heuristics. Table 4.9 gives the total number of migrations observed under the aberrant scenario, with and without budget enforcement. We continue to consider tasks  $T_1$  and  $T_2$  described earlier in Section 4.4.1. We make two observations.

**Observation 34.** *Affinity-aware GPU assignment helps maintain affinity.*

Migration Type	C-EDF				C-RM			
	No Budgeting		Budgeting		No Budgeting		Budgeting	
	$T_1$	$T_2$	$T_1$	$T_2$	$T_1$	$T_2$	$T_1$	$T_2$
No Migration	11,887	2,382	8,866	2,374	11,255	1,826	8,586	1,915
Near	110	11	130	8	374	245	386	142
Far	0	4	0	13	368	326	173	340
Total	11,997	2,397	8,996	2,395	11,997	2,397	9,145	2,397

Table 4.9: Migration frequency for  $T_1$  and  $T_2$ .

Under all scenarios, we see that tasks are significantly more likely to be assigned the GPU with which they have affinity. For instance, task  $T_1$ , under C-EDF with no budget enforcement, maintained affinity 11,887 times, migrated to a neighboring GPU 110 times, and never migrated to a distant GPU. Similar trends are observed for all tasks. Without an affinity-aware method, *migrations would be more frequent than reassignment*, since any task would have a 75% chance (with a GPU-cluster size of four) of being assigned to a different GPU.

**Observation 35.** *GPUSync is more successful at maintaining affinity under the C-EDF configuration.*

Observe in Table 4.9 that migrations are more frequent under C-RM than C-EDF. Indeed, distant migrations are practically eliminated under C-EDF. This behavior is attributable to the decreased accuracy of the Cost Predictor under C-RM (recall Observation 32). Inaccurate estimates can cause migrations to seem faster than waiting for the GPU for which the job has affinity. This observation demonstrates the importance of the Cost Predictor with respect to the runtime behavior.

GPUSync can effectively constrain GPU utilization through budget enforcement techniques, produce relatively accurate predictions of GPU migration costs, and effectively employ GPU allocation heuristics that significantly reduce the frequency of GPU migrations.

This concludes our focused evaluation of the elemental components of GPUSync. We now examine overall runtime performance.

#### 4.4.2 Feature-Tracking Use-Case

We now describe the vision-related experiments mentioned earlier. In these experiments, we adapted a freely-available CUDA-based feature tracking program to GPUSync on LITMUS<sup>RT</sup> by Kim *et al.* (2009). Feature tracking detects interesting features, such as lines and corners, in a video image and tracks the

movement of detected features through time. Feature tracking can be used to deduce the three-dimensional structure of environment and movement of the camera. Feature tracking can be an important element of a pipeline of computer vision algorithms used by ADAS and autonomous automotive systems that sense and monitor the environment. The tracker represents a scheduling challenge since it utilizes both CPUs and GPUs to carry out its computations. Though feature tracking is only one GPGPU application, its image-processing operations are emblematic of many others.

We stressed our evaluation platform by applying feature tracking to thirty independent video streams simultaneously. Each video stream was handled by one task, with each frame being processed by one job. The video streams were assigned different execution-rate requirements: two high-rate streams ran at 30 frames per second (FPS), ten medium-rate streams ran at 20FPS, and eighteen low-rate streams ran at 10FPS. The high- and medium-rate streams operated on video frames with a resolution of 320x240 pixels. Each job of these streams had roughly 1MB of combined input and output data and a state size of about 6.5MB. The low-rate streams processed larger video frames with a resolution of 640x480. A low-rate stream required four times as much memory as a higher-rate stream. Video frames were preloaded into memory in order to avoid disk latencies (such latencies would be non-existent with real video cameras). All data was page-locked in system memory to facilitate fast and deterministic memory operations.

We tested the same configurations as in the prior runtime experiments, with the addition of another GPUSync configuration under C-EDF scheduling: the use of priority-ordered engine locks. The token count was also reduced to  $\rho = 2$  for all configurations—we found that this configuration worked best for our particular computer vision workload.<sup>18</sup> The video streams were scheduled on three different GPU clustering configurations: eight GPU partitions ( $g = 1$ ), four small GPU clusters of two GPUs ( $g = 2$ ), and two large GPU clusters of four GPUs ( $g = 4$ ). CPUs were organized in two clusters of six. We focused our attention on platform configurations where GPU clusters *were not* shared by CPUs of different CPU clusters (recall that such configurations performed exceedingly poorly in our schedulability experiments). Each CPU cluster was associated with the same number of GPU clusters (and hence, the same number of GPUs). We tested the clustered configurations with both peer-to-peer and system memory migration. Tasks were partitioned evenly among the CPU and GPU clusters. Our video-stream workload was scheduled under each configuration for 120 seconds and measurements were taken.

---

<sup>18</sup>In Chapter 5, we find that  $\rho = 2$  does not necessarily lead to the best runtime performance on our evaluation platform in general. Indeed, greater values of  $\rho$  may be *necessary* to achieving predictable runtime performance.

We analyze each GPUSync configuration by inspecting the response time of each job. We deem a configuration as resulting in an unschedulable system if any task consistently exhibits growth in response time, since this is a sign of unbounded deadline tardiness. We assume a schedulable system with bounded deadline tardiness, otherwise.

Figure 4.20 plots the CDFs of *normalized* job response times for the platform configurations that employed FIFO-ordered engine locks and C-EDF scheduling. We normalize job response times by dividing the observed response time by the period of the associated task; we do so that we may combine job response times into a single CDF. Figures 4.21 and 4.22 give the same type of plots for platforms that employed priority-ordered engine locks, under C-EDF and C-RM, respectively. Unschedulable configurations are denoted by dashed lines. We clip the plots in order to show more detail for shorter response times. Clipped curves are annotated with the value (in parentheses) where the associated CDF reaches 1.0 (*i.e.*, the observed worst-case response time). We make several observations from these results.

**Observation 36.** *GPUSync can be used to achieve predictable real-time performance for GPGPU applications.*

This experiment provides our first look at the *overall* runtime performance of GPUSync. The CDFs in Figures 4.20, 4.21, and 4.22 demonstrate that GPUSync can be used to realize real-time predictability for GPGPU applications. For instance, examine the CDF for GPU clusters of size two in Figure 4.20 (curve 2). We see that all jobs had a normalized response time less than approximately 180% of period. We take this as an indicator of bounded deadline tardiness. CDFs that reflect even better real-time performance can be found in Figures 4.21 and 4.22. This result provides evidence supporting a central tenant of this dissertation: real-time scheduling and synchronization techniques can be applied to GPUs to realize predictable real-time performance.

**Observation 37.** *Priority-ordered queues can provide improved observed response times.*

This can be observed by comparing the curves in Figure 4.20 to those of Figures 4.21 and 4.22. For example, in Figure 4.20, the probability that a job under partitioned GPU scheduling (curve 1) had normalized response time less than 200% was approximately 90%. Compare this to Figure 4.21, where 100% of jobs had a normalized response time less than 200%. We can observe less dramatic differences in performance for the clustered GPU configurations that used peer-to-peer migration (curves 2 and 4).

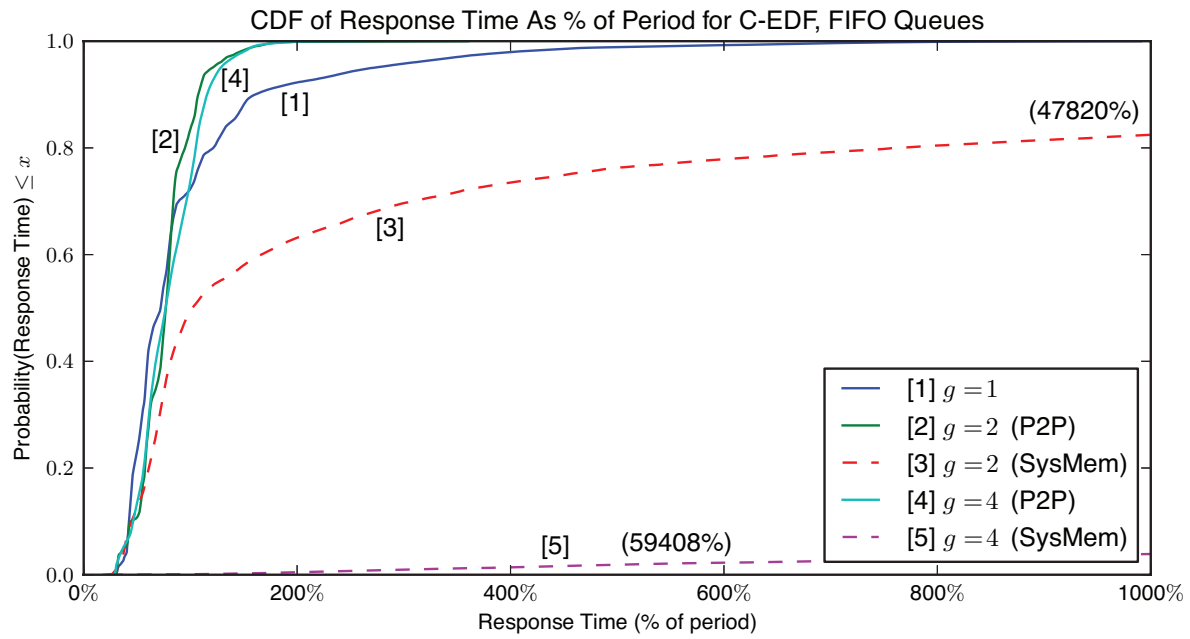


Figure 4.20: CDF of job response time as percent of period for C-EDF with FIFO-ordered engine locks.

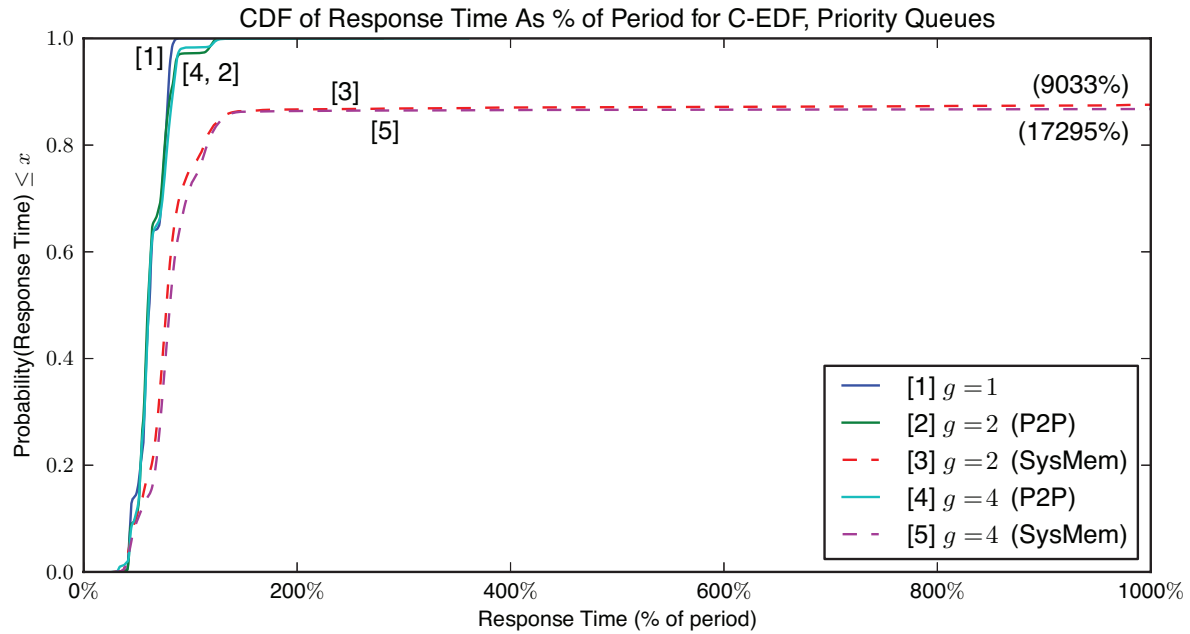


Figure 4.21: CDF of job response time as percent of period for C-EDF with priority-ordered engine locks.

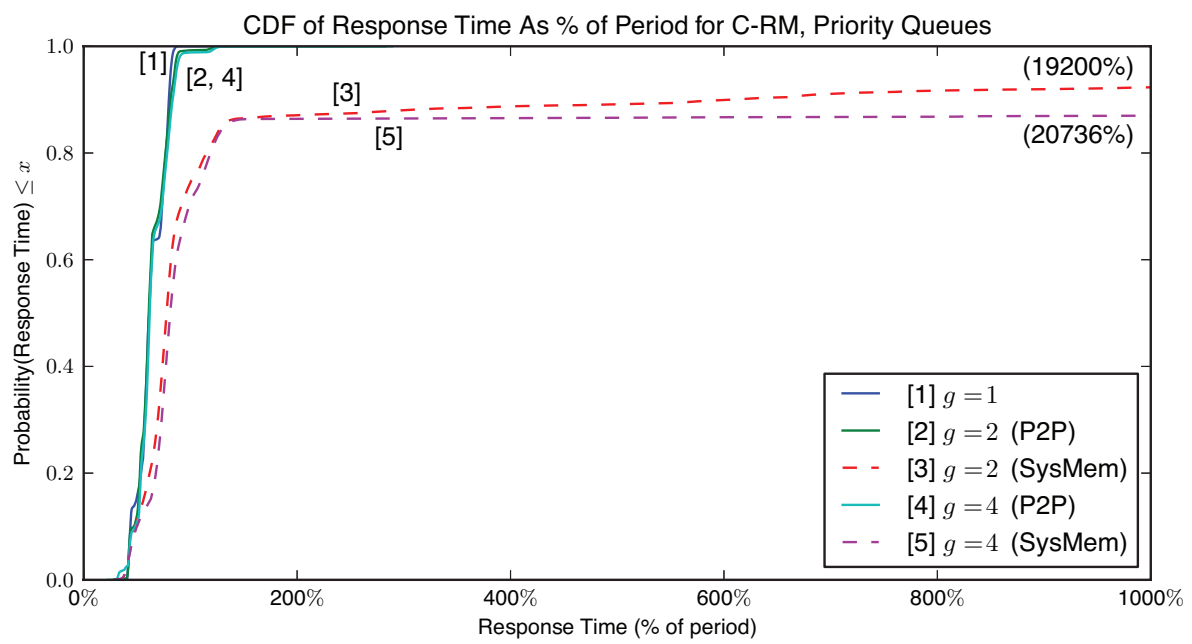


Figure 4.22: CDF of job response time as percent of period for C-RM with priority-ordered engine locks.

**Observation 38.** *Clustered GPU management can outperform a partitioned approach.*

This can be observed in curves 2 and 4 of Figure 4.20 for C-EDF scheduling with FIFO-ordered engine locks. The CDFs for the clustered GPU configurations with peer-to-peer migration generally lie above the CDFs for the other configurations (particularly, curve 1 for partitioned GPU scheduling). This indicates that a job under clustered GPU scheduling with peer-to-peer migration is more likely to have a shorter normalized response time than jobs in the other FIFO-ordered configurations. Despite the extra load imposed by memory migrations, clustered GPU approaches can still outperform a partitioned GPU approach. This is a positive result in light of the benefits FIFO ordering can have in schedulability analysis. Clustering may provide a viable FIFO-based alternative to a poorer performing partitioned approach.

**Observation 39.** *Small GPU clusters may be preferable over large GPU clusters.*

In Figures 4.20, 4.21, and 4.22, we observe that small GPU clusters (curves 2 and 3) generally yield shorter normalized response times than large GPU clusters (curves 4 and 5) in this experiment. For example, in Figure 4.20, the curve for the platform configuration with a GPU cluster size of two with peer-to-peer migrations (curve 2) lies above the curve for a GPU cluster size of four with peer-to-peer migrations (curve 4). This difference is partly attributable to the higher cost of migration between distant GPUs in larger clusters, which we observed in Section 4.3.1.2 (Figures 4.8 and 4.9). However, we see that the smaller GPU cluster configurations still outperform the larger GPU cluster configurations when system memory migrations are used. We cannot attribute this difference in performance to migration distance, since GPUs in both configurations have the same migration distance to system memory. Instead, we attribute this difference to an increased chance of migration in larger GPU clusters.

**Observation 40.** *GPU state migration through system memory is often too costly.*

We observe in Figures 4.20, 4.21, and 4.22, that no configuration that used system memory migration resulted in a schedulable system in our experiment. Indeed, we observe that partitioned GPU management is preferable to all such system-memory-migration configurations. This is clearly reflected by curves 3 and 5 in each figure for GPU clusters of size two and four, respectively, where the observed worst-case response times are over 9,000% of task period. Furthermore, system memory migrations are even more costly for larger cluster sizes on account of the increased frequency of migrations between GPUs, thereby further increasing response times.

**Observation 41.** *GPUSync can be used effectively with both C-EDF and C-RM schedulers.*

We compare the figures for the C-EDF and C-RM platform configurations that used priority-ordered engine locks to contrast the performance of these schedulers. We ignore GPU cluster configurations that use system memory migration, since our workload is clearly unschedulable (Observation 40). Compare the remaining corresponding curves in Figures 4.21 (C-EDF) and 4.22 (C-RM)—the corresponding CDFs are virtually indistinguishable. For example, compare the curves for GPU clusters of size two that use peer-to-peer migration (curves 2): the curves are nearly identical. The same holds true of the corresponding curves for partitioned GPUs and the GPU clusters of size four.

GPUSync is meant to provide a configurable real-time GPU scheduling framework. We have mainly focused on configurability as it relates to the number of GPU tokens and engine lock queues. However, this configurability also extends to support for any JLFP CPU scheduler. Our observation here helps demonstrate our success at meeting this design goal. We see that GPUSync can be just as effective under C-EDF scheduling as it is under C-RM scheduling.

This concludes our first evaluation GPUSync’s runtime performance. We direct the reader towards Chapter 5 for additional experiments and in-depth analysis of GPUSync’s runtime performance when it used to support graph-based real-time workloads.

## 4.5 Conclusion

In this chapter, we have evaluated both the analytical and runtime aspects of GPUSync. We have examined the overheads associated with GPGPU computing in depth. Through carefully crafted experiments, we have quantified and characterized the runtime properties of GPU interrupt processing and DMA operations. We have also measured the degree to which DMA operations may increase the execution time of tasks executing on CPUs due to memory bus contention. We incorporated these overheads, among others, into an analytical model of our evaluation platform.

We also presented blocking analysis for various GPUSync configurations. We provided this analysis in varying degrees of granularity. We began with “coarse-grain” analysis in order to study the overall characteristics of different GPUSync configurations. We then developed more fine-grain analysis for use in schedulability experiments. In this analysis, we also discussed the tradeoffs between various GPUSync configurations.



We evaluated the analytical performance of GPUSync through a set of large scale schedulability experiments, where we tested over 2.8 billion task sets for schedulability. These experiments required over 85,000 CPU hours to complete on a university compute cluster. Through these experiments, we identified the most promising GPUSync configurations.

We evaluated the runtime properties of GPUSync through both targeted experiments and a computer vision use-case study. The results of our targeted experiments show that GPUSync’s affinity-aware heuristics are effective at reducing GPU migration, and that these techniques can practically eliminate the most costly types of migrations. These targeted experiments also showed that GPUSync’s budget enforcement policies help prevent aberrant tasks from exceeding provisioned GPU execution times and mitigate the negative effects on other tasks when they do. Finally, in the computer vision use-case study, we found that GPUSync can effectively schedule real-time work on GPUs. Moreover, we saw that in some configurations, clustered GPU scheduling can outperform more conventional partitioned GPU approaches.

## CHAPTER 5: GRAPH SCHEDULING AND GPUS<sup>1</sup>

In Chapter 4, we used GPUSync to arbitrate access to GPU resources among tasks that follow the sporadic task model, whereby each real-time task is represented by a single thread of execution. In this chapter, we apply GPUSync towards the graph-based PGM task model described earlier in Section 2.1.3. We begin by providing our motivation for supporting a graph-based task model, which we draw from recent trends in real-world software architectures and application constraints. We then describe PGM<sup>RT</sup>, a middleware library we developed to support real-time task sets derived from PGM graphs. PGM<sup>RT</sup> integrates tightly with the LITMUS<sup>RT</sup> kernel (which we also modified to support PGM-derived real-time task sets) to minimize system overheads. However, PGM<sup>RT</sup> remains portable to POSIX-compliant operating systems. Next, we discuss the newly developed open standard, OpenVX<sup>TM</sup>, which is designed to support computer vision applications (Khronos Group, 2014c).<sup>2</sup> OpenVX uses a graph-based software architecture designed to enable efficient computation on heterogeneous computing platforms, including those that use accelerators like GPUs. We examine assumptions made by the designers of OpenVX that conflict with our real-time task model. We then discuss VisionWorks<sup>®</sup>, an OpenVX implementation by NVIDIA (Brill and Albuz, 2014).<sup>3</sup> With support from NVIDIA, we adapted an alpha version of VisionWorks to run atop PGM<sup>RT</sup>, GPUSync, and LITMUS<sup>RT</sup>. We describe several challenges we faced in this effort, along with our solutions. We then present the results from a runtime evaluation of our modified version of VisionWorks under several configurations of GPUSync. We compare our GPUSync configurations against two purely Linux-based configurations, as well as a LITMUS<sup>RT</sup> configuration *without* GPUSync. Our results demonstrate clear benefits from GPUSync. We conclude this chapter with a summary of our efforts and experimental results.

---

<sup>1</sup> Portions of this chapter previously appeared in conference proceedings. The original citation is as follows: Elliott, G., Kim, N., Liu, C., and Anderson, J. (2014). Minimizing response times of automotive dataflows on multicore. In *Proceedings of the 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10.

<sup>2</sup>OpenVX is a trademark of the Khronos Group Inc.

<sup>3</sup>VisionWorks is a registered trademark of the NVIDIA Corp.

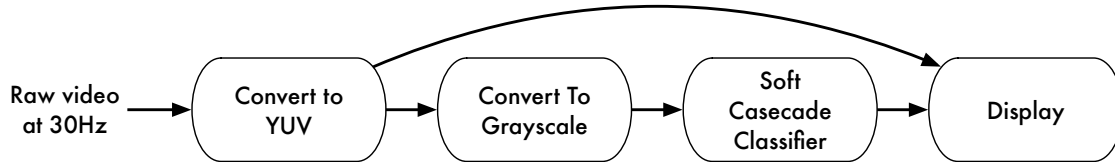


Figure 5.1: Dataflow graph of a simple pedestrian detector application.

## 5.1 Motivation for Graph-Based Task Models

Graph-based software architectures, often referred to as *dataflow* architectures, are common to software applications that process continual streams of data or events. In such architectures, vertices represent sequential code segments that operate upon data, and edges express the flow of data among vertices. The flexibility offered by such an architecture’s inherent modularity promotes code reuse and parallel development. Also, these architectures naturally support concurrency, since parallelism can be explicitly described by the graph structure. These characteristics have made dataflow architectures popular in multimedia technologies (Khronos Group, 2005; Taymans *et al.*, 2013) and the emerging field of computational photography (Adams *et al.*, 2010; NVIDIA, 2013). Dataflow architectures are also prevalent in the sensor-processing components in prototypes of advanced automotive systems, for both driver-assisted and autonomous driving (*e.g.*, Miller *et al.* (2009); Urmson *et al.* (2009); Wei *et al.* (2013)). While many domains with dataflow architectures have timing requirements, the automotive case is set apart since timing violations may result in loss of life or property.

Figure 5.1 depicts a dataflow graph of a simple pedestrian detection application that could be used in an automotive application. We describe these nodes from left to right. A video camera feeds the source of the graph with video frames at 30Hz (or 30FPS). The first node converts raw camera data into the common YUV color image format. The second node extracts the “Y” component of each pixel from the YUV image, producing a grayscale image. (Computer vision algorithms often operate only on grayscale images.) The third node performs pedestrian detection computations and produces a list of the locations of detected pedestrians. In this case, the node uses a common “soft cascade classifier” (Bourdev and Brandt, 2005) to detect pedestrians. Finally, the last node displays an overlay of detected pedestrians over the original color image.

We may shoehorn our pedestrian detection application into a single implicit deadline sporadic task. Here, we give such a task a period and relative deadline of  $33\frac{1}{3}$ ms to match the period of the video camera. Each

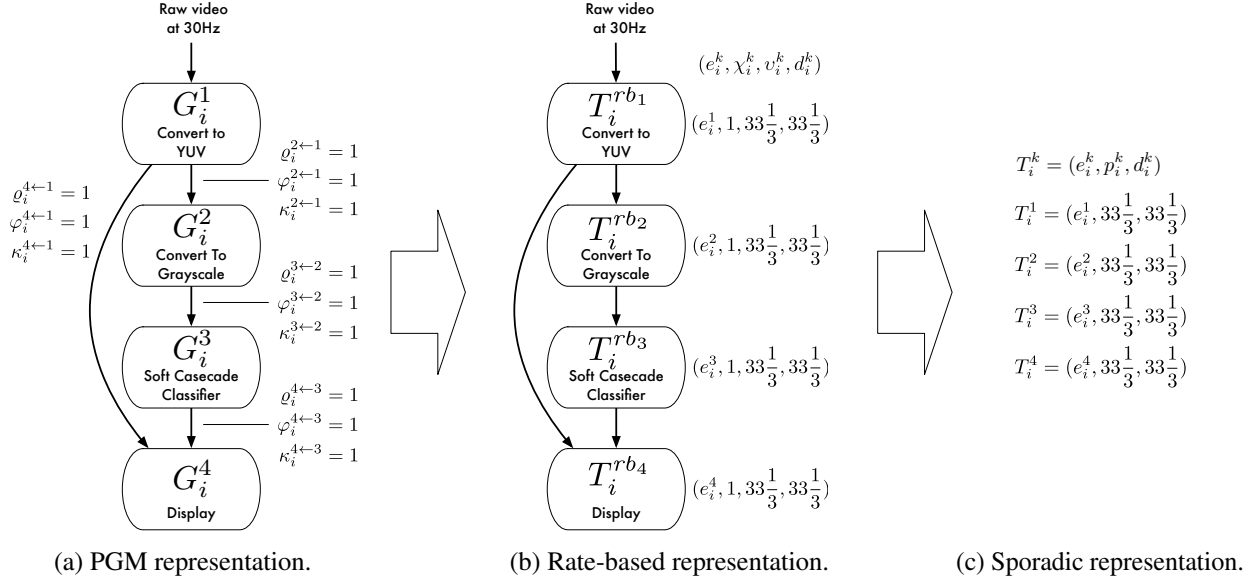


Figure 5.2: Transformation of a PGM-specified graph for a pedestrian detection application to sporadic tasks.

job of this task executes the dataflow graph, end-to-end, once per job. This technique can be applied to any graph by executing graphs nodes in a topological order. However, this approach prevents us from exploiting parallelism in two ways. First, we cannot exploit parallelism expressed by parallel branches (or forks) in a graph since we serialize all graph nodes. Second, we cannot execute nodes in a pipeline, since the graph is executed end-to-end within each job. There is another significant drawback to this shoehorned approach: the combined execution time of graph nodes may exceed the period of the task. Such a task is intrinsically unschedulable without parallelism.

In Section 2.1.3, we described a process for transforming a dataflow graph described by PGM into a set of sporadic tasks. Figure 5.2 depicts such a transformation for our pedestrian detection application. We describe this transformation in more detail. Figure 5.2(a) gives a PGM-specification for the pedestrian detection dataflow graph. Here, each non-sink node produces one token ( $\rho_i^{k \leftarrow j} = 1$ ) for each of its consumers. Similarly, each non-source node consumes one token ( $\kappa_i^{k \leftarrow j} = 1$ ) from each of its producers, as tokens become available ( $\phi_i^{k \leftarrow j} = 1$ ). Given an input video rate of 30Hz, the rate-based task for the source node,  $T_i^{rb1}$ , is released once ( $\chi_i^{rb1} = 1$ ) every  $33\frac{1}{3}$ ms ( $v_i^{rb1} = 33\frac{1}{3}$ ms). The remaining nodes have the same rate-based specification since  $\rho_i^{k \leftarrow j} = \kappa_i^{k \leftarrow j}$  (see Equations (2.8)–(2.11)). This is depicted in Figure 5.2(b). Finally, Figure 5.2(c) gives the final transformation to implicit-deadline sporadic tasks, where  $d_{i,j}^k = p_{i,j}^k = 33\frac{1}{3}$ ms.

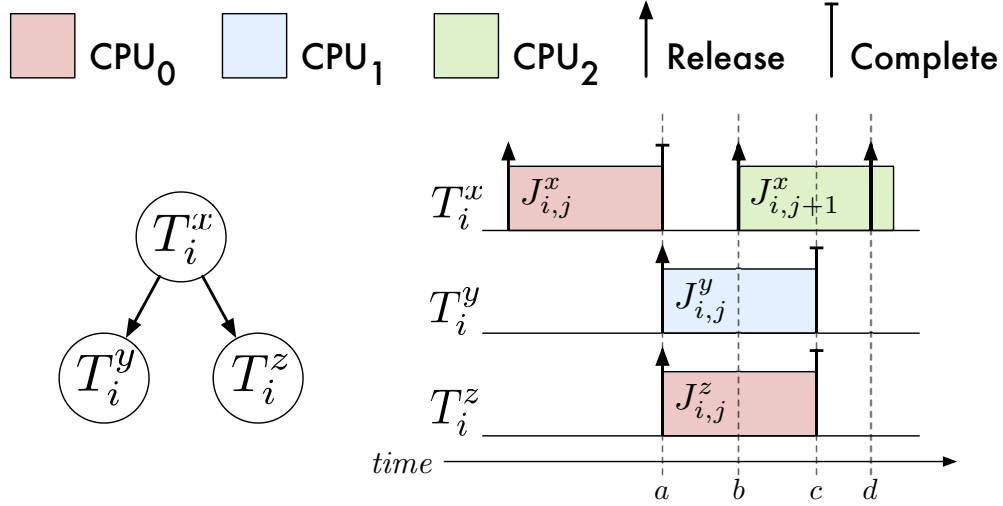


Figure 5.3: Parallel execution of graph nodes.

Ideally, any node with satisfied constraints should be eligible for scheduling to allow the parallel execution and pipelining of nodes. This would allow three types of parallelism: inter-node, intra-node, and pipeline parallelism. Consider the graph and schedule depicted in Figure 5.3. Here, the completion of job  $J_{i,j}^x$  at time  $a$  releases jobs  $J_{i,j}^y$  and  $J_{i,j}^z$ . These released jobs execute in parallel during the time interval  $[a, c]$ . This is an example of inter-node parallelism, since two nodes released by a shared parent may execute simultaneously. Job  $J_{i,j+1}^x$  is released at time  $b$ . It is scheduled in parallel with jobs  $J_{i,j}^y$  and  $J_{i,j}^z$  during the time interval  $[b, c]$ . This is an example of pipeline parallelism, since  $J_{i,j+1}^x$  may execute before the jobs released by  $J_{i,j}^x$  complete. Job  $J_{i,j+2}^x$  is released at time  $d$ . If  $J_{i,j+2}^x$  were scheduled before the completion of  $J_{i,j+1}^x$ , then this would be an example of intra-node parallelism. However, the schedule depicted in Figure 5.3 observes job precedence constraints; *i.e.*, two instances of the *same* node may not execute simultaneously.

The transformation of a PGM-specified graph into a set of sporadic tasks gives us *most* of the parallelism we seek. Although it would be desirable to support intra-node parallelism, it may be challenging to realize in implementation, since it may require dynamic thread creation or *preemptive* work assignment to threads in a pool of worker threads. Moreover, intra-node parallelism may also require us to re-order or buffer node outputs, since the  $(j+1)^{st}$  invocation of a node may complete before its  $j^{th}$  invocation. From a practical perspective, we feel the loss of intra-node parallelism is acceptable.

## 5.2 PGM<sup>RT</sup>

In this section, we describe PGM<sup>RT</sup>, a middleware library that we developed to support PGM-derived real-time task sets. PGM<sup>RT</sup> is responsible for transmitting tokens (and, optionally, data) among the nodes and enforcing token constraints.<sup>4</sup> PGM<sup>RT</sup> may be configured to integrate with LITMUS<sup>RT</sup> in order to reduce token transmission overheads and ensure predictable real-time execution. However, PGM<sup>RT</sup> may also be configured to support other POSIX-compliant platforms at the expense of greater overheads and priority inversions.

We begin by describing the underlying mechanisms of PGM<sup>RT</sup>, specifically, graph management and token transmission. We then address issues relating to proper real-time scheduling of tasks running under our PGM-derived sporadic task model on LITMUS<sup>RT</sup>.

### 5.2.1 Graphs, Nodes, and Edges

Each graph is identified by a unique name and path, similar to a UNIX named pipe. Applications use PGM<sup>RT</sup>'s API to create new graphs described by nodes and edges. Real-time tasks, as unique threads of execution within the same address space or separate processes, use PGM<sup>RT</sup>'s API to access information about a named graph and claim/bind to a node and its edges. PGM<sup>RT</sup> uses a plugin-based architecture to support different methods for transmitting tokens (and, optionally data) among tasks.

### 5.2.2 Precedence Constraints and Token Transmission

Non-source nodes have two types of precedence constraints: job and token constraints. Job constraints are satisfied in PGM<sup>RT</sup>, since a single thread binds to each node—jobs are naturally serialized by this thread. Regarding token constraints, consumers block (suspend execution) whenever they lack the requisite tokens. Producers must have a mechanism to signal consumers of new tokens. The appropriate underlying IPC mechanism depends upon how tokens are used: tokens may be *event-signaling* or *data-passing*. A single node may use a mix of event-signaling and data-passing tokens, as appropriate. Regardless of the underlying IPC, nodes produce and consume tokens using a common API.

**Event-Signaling Tokens.** With event-signaling tokens, token production and consumption is realized through increment/decrement operations on per-edge token counters, similar to counting semaphores. To facilitate

---

<sup>4</sup>We distribute PGM<sup>RT</sup> as open source under the Revised BSD license. Source code is currently available at [www.github.com/GElliott/pgm](http://www.github.com/GElliott/pgm).

IPC, token counters may be stored in POSIX shared memory that is mapped into each PGM<sup>RT</sup> application. Thus, a single graph may be made up of multiple coordinated processes.

Although tokens do not transmit data implicitly, tokens can coordinate data sharing in application-level logic. For example, a token may signal the availability of new data in an out-of-band queue (*i.e.*, a data structure outside the purview of PGM<sup>RT</sup>) shared between two nodes. Signaling is achieved via a monitor synchronization primitive (which can also be stored in shared memory to facilitate inter-process communication). For POSIX-compliant operating systems, this monitor is realized by a POSIX (pthread) condition variable, one per consumer. A consumer blocks on its condition variable if it does not have requisite tokens on every edge. A producer signals the condition variable whenever it is the last producer to satisfy all of its consumer's token constraints.

The use of pthread condition variables has the drawback that threads that utilize the synchronization primitive must first acquire a pthread mutex. This suspension-based mutex can be problematic in a real-time setting for three reasons. First, it can introduce heavy overheads, due to context switching, with respect to very short critical sections in PGM<sup>RT</sup> (*e.g.*, the increment/decrement of a handful of token counters). Such overheads may be pessimistically accounted for in real-time analysis, but it is desirable to avoid them altogether. Second, suspensions are difficult to model under some methods of real-time analysis (*e.g.*, *s*-oblivious analysis). Finally, an operating system may offer little or no support for real-time priority inheritance for pthread mutexes. As an alternative to pthread-based monitor, PGM<sup>RT</sup> also offers a FIFO-ordered spinlock-based monitor built upon Linux's "fast user-space mutex" (or "futex") API. Its use can eliminate costly context switches and problematic suspensions. Furthermore, the duration of priority inversions is bounded since spinning tasks wait non-preemptively. On non-LITMUS<sup>RT</sup> platforms, non-preemptive waiting is achieved by disabling CPU interrupts from the user-space (*e.g.*, `sti/cli/pushf/popf` instructions on x86 processors). On LITMUS<sup>RT</sup>, PGM<sup>RT</sup> uses LITMUS<sup>RT</sup>'s special support for non-preemptive code sections. Here, an application enters and exits non-preemptive sections by merely writing to a variable shared by the application and LITMUS<sup>RT</sup>.

**Data-Passing Tokens.** With data-passing tokens, each byte-sized token is interpreted as a byte of data. Byte-ordering is preserved through FIFO-ordered token consumption. Data-passing can be achieved through a variety of IPC channels. PGM<sup>RT</sup> supports PGM<sup>RT</sup>-provided ring buffers, named pipes (FIFOs), message queues (MQs), and stream sockets (*e.g.*, TCP)—all IPCs are POSIX-standard/compatible. We classify these

mechanisms collectively as IPC channels. One channel is created for each edge. With the exception of the ring-buffer IPC,<sup>5</sup> consumers block on the set of channels, represented as a list of file descriptors, from each inbound edge using `select()`.<sup>6</sup> The operating system wakes the consumer whenever data arrives on a previously empty channel. Consumers `read()/recv()` tokens from the channel once tokens are available on all edges. Under PGM<sup>RT</sup>, all read/write operations are non-blocking in order to avoid excessive thread suspensions. Under non-blocking writes, producers continually write data from within a loop until all data has been written. Consumers similarly loop while reading. As a fail-safe mechanism, consumers suspend through `select()` if inbound data is exhausted before the requisite number of bytes have been read. Due to the general lack of introspective capabilities with the above IPC mechanisms (specifically, the inability to query the IPC channel regarding the amount of available data), PGM consumer thresholds greater than the number of tokens consumed per node invocation are not easily supported. However, PGM<sup>RT</sup> offers a solution to this problem that we discuss next.

The use of `select()` for data-passing tokens can introduce additional thread suspensions since `select()` wakes a blocked thread when data becomes available on any *one* channel. Thus, a consumer waiting for data on all inbound edges must loop on `select()` until data arrives on all inbound edges.<sup>7</sup> To avoid this, PGM<sup>RT</sup> offers “fast” variants of FIFO- and MQ-based channels (and PGM<sup>RT</sup> ring buffers are only available in this flavor), where the underlying channel IPC is *wrapped* with event-signaling tokens. Here, the availability of data is tracked by event-signaling tokens, with each token corresponding to one byte of data. As with plain event-signaling tokens, consumers block on a monitor, and are awoken by the last producer to satisfy all token constraints. Producers only transmit event-signaling tokens after they have written the produced data to the associated channel. Thus, consumers using the fast channels avoid repeated suspensions while looping on `select()`. Moreover, support for PGM consumer thresholds greater than the number of tokens consumed per node invocation is trivialized by event-signaling token counters. Thus, the fast channel variants also support PGM’s consumer thresholds. There is one limitation to using event-signaling tokens in this context: a reliance upon shared memory. As a result, PGM<sup>RT</sup> does not offer a fast variant of stream socket

---

<sup>5</sup>Ring-buffers use the “fast” method described next.

<sup>6</sup>The use of `select()` with MQs is not strictly POSIX-compliant, but such use is commonly supported.

<sup>7</sup>An “all-or-nothing” variant of `select()` could be used to address this issue. However, we are unaware of any OS that supports such an API.



channels, since producers and consumers connected by these channels are expected to reside on different computers in a distributed system.

### 5.2.3 Real-Time Concerns

PGM<sup>RT</sup> described as above can be used with general-purpose schedulers. However, additional enhancements are required to ensure predictable real-time behavior. These relate to predictable token signaling and proper deadline assignment. We describe how we address these problems when PGM<sup>RT</sup> runs atop LITMUS<sup>RT</sup>.

**Early Releasing and Deadline Shifting.** Under deadline-based schedulers, the technique of *early releasing* allows a job to be scheduled prior to its release time, provided that the job’s absolute deadline is computed from the normal (*i.e.*, “non-early”) release time. Under LITMUS<sup>RT</sup>, the jobs of all tasks associated with non-source nodes are released early. However, early-released jobs must still observe token constraints.

A job’s absolute deadline is computed as  $d_i$  time units after its (non-early) release. LITMUS<sup>RT</sup> employs high-resolution timers to track the minimum separation time between releases. However, recall from Section 2.1.3 that the release time of a non-source node can be no less than the time instant the job’s token constraints are satisfied. Thus, the absolute deadline for each job must be computed on-the-fly. Immediately before a consumer blocks for tokens, it sets a “token-wait” flag stored in memory shared by user-space and the kernel. The kernel checks this flag whenever a real-time task is awoken from a sleeping state. If set, and the current time is later than the release time dictated by the sporadic task model, the kernel automatically computes an adjusted release and absolute deadline for the job and clears the flag. This computation requires the current time to approximate the arrival time of the last token—this is ensured by boosting the priority of a producer while it signals consumers. We discuss this next.

**Priority-Boosting of Producers.** To ensure properly computed deadlines, we boost the priority of a producer while it is signaling a sequence of consumers. Moreover, in cases where a graph spans multiple processor clusters, boosting is necessary to avoid leaving processors in remote clusters idle. Boosting is achieved through a lazy process: the priority of a producer is boosted only when the scheduler attempts to preempt it. A producer informs LITMUS<sup>RT</sup> of when it is signaling consumers through a “token-sending” flag stored in memory shared by user-space and the kernel. Once all consumers have been signaled, a producer clears the “token-sending” flag and triggers LITMUS<sup>RT</sup> to “unboost” and reschedule the producer. Priority inversions due to boosting should be accounted for in real-time analysis.

We now discuss applications where we may use PGM<sup>RT</sup> to help realize real-time graph scheduling, especially those that use GPUs.

### 5.3 OpenVX

OpenVX is a newly ratified standard API for developing computer vision applications for heterogeneous computing platforms. The API provides the programmer with a set of basic operations, or *primitives*, commonly used in computer vision algorithms.<sup>8</sup> The programmer may supplement the standard set of OpenVX primitives with their own or with those provided by third-party libraries. Each primitive has a well-defined set of inputs and outputs. The implementation of a primitive is defined by the particular implementation of the OpenVX standard. Thus, a given primitive may use a GPU in one OpenVX implementation and a specialized DSP (e.g., CongniVue’s G2-APEX or Renesas’ IMP-X4) or mere CPUs in another. OpenVX also defines a set of *data objects*. Types of data objects include simple data structures such as scalars, arrays, matrices, and images. There are also higher-level data objects common to computer vision algorithms—these include histograms, image pyramids, and lookup tables.<sup>9</sup> The programmer constructs a computer vision algorithm by instantiating primitives as *nodes* and data objects as *parameters*. The programmer binds parameters to node inputs and outputs. Since each node may use a mix of the processing elements of a heterogeneous platform, a single graph may execute across CPUs, GPUs, DSPs, etc.

Node dependencies (*i.e.*, edges) are not explicitly provided by the programmer. Rather, the structure of a graph is derived from how parameters are bound to nodes. We demonstrate this with an example. Figure 5.4(a) gives the relevant code fragments for creating an OpenVX graph for pedestrian detection. The data objects `imageRaw` and `detected` represent the input and output of the graph, respectively. The data objects `imageIYUV` and `imageGray` store an image in color and grayscale formats, respectively. At line 12, the code creates a color-conversion node, `convertToIYUV`. The function that creates this node, `vxColorConvertNode()`, takes `imageRaw` and `imageIYUV` as input and output parameters, respectively. Whenever the node represented by `convertToIYUV` is executed, the contents of `imageRaw` is processed by the color-conversion primitive, and the resulting image is stored in `convertToIYUV`. Similarly, the node `convertToGray` converts the color image into a grayscale image. The grayscale image is processed

---

<sup>8</sup>The OpenVX specification calls these basic operations “kernels.” However, we opt to avoid this term since we must already differentiate between GPU and OS kernels.

<sup>9</sup>An image pyramid stores multiple copies of the same image. Each copy has a different resolution or scale.

```

1 vx_image imageRaw; // graph input : an image
2 vx_array detected; // graph output : a list of detected pedestrians
3 ...
4 // instantiate a graph
5 vx_graph pedDetector = vxCreateGraph(...);
6 ...
7 // instantiate additional parameters
8 vx_image imageIYUV = vxCreateVirtualImage(pedDetector, ...);
9 vx_image imageGray = vxCreateVirtualImage(pedDetector, ...);
10 ...
11 // instantiate primitives as nodes
12 vx_node convertToIYUV = vxColorConvertNode(pedDetector, imageRaw, imageIYUV);
13 vx_node convertToGray = vxChannelExtractNode(pedDetector, imageIYUV,
14                                             VX_CHANNEL_Y, imageGray);
15 vx_node detectPeds = mySoftCascadeNode(pedDetector, imageGray, detected, ...);
16 ...
17 vxProcessGraph(pedDetector); // execute the graph end-to-end

```

(a) OpenVX code for constructing a graph.



(b) Bindings of data object parameters to nodes.



(c) Derived graph structure.

Figure 5.4: Construction of a graph in OpenVX for pedestrian detection.

by a user-provided node created by the function `mySoftCascadeNode()`, which writes a list of detected pedestrians to `detected`.<sup>10</sup> Figure 5.4(b) depicts the bindings of parameters to nodes. Figure 5.4(c) depicts the derived structure of this graph.

OpenVX defines a simple execution model. From Section 2.8.5 of the OpenVX standard:

*[A constructed graph] may be scheduled multiple times but only executes sequentially with respect to itself.*

Moreover:

*[Simultaneously executed graphs] do not have a defined behavior and may execute in parallel or in series based on the behavior of the vendor's implementation.*

<sup>10</sup>The OpenVX standard does not currently specify a primitive for object detection, so the user must provide their own or use one from a third party.

This execution model simplifies the OpenVX API and its implementation. Also, this simplicity is partly motivated by the variety of heterogeneous platforms upon which OpenVX applications are meant to run. The API must work well for simple processors such as ASICs as well as modern CPUs. Furthermore, the simple execution model enables interesting opportunities for optimization. For example, an entire graph could be transformed into gate-level logic or a single GPU kernel and executed entirely on an FPGA or GPU, respectively. However, OpenVX’s execution model has four significant implications on real-time scheduling. First, the specification has no notion of a periodic or sporadic task. Second, the specification only allows the programmer to control when the root node of a graph is ready for execution, not when *internal* nodes are ready. Third, the specification does not define a threading model for graph execution. The intent of the standard is to allow the OpenVX implementation to be tailored to particular heterogeneous platforms. However, it provides no mechanism by which to control the number of threads used to execute a graph or the priority of these threads. Finally, the specification requires a graph to execute end-to-end before it may be executed again. This makes pipelining impossible.<sup>11</sup>

Given these limitations, how can we execute OpenVX graphs under a sporadic task model? As we discussed in Section 5.1, we may assign a single graph to a single sporadic real-time task. A job of this task would execute the nodes of the graph serially in topological order. Of course, we miss opportunities to take advantage of inherent graph parallelism with this approach. We present a better solution in the next section.

## 5.4 Adding Real-Time Support to VisionWorks

VisionWorks is an implementation of OpenVX developed by NVIDIA. Many of the OpenVX primitives are implemented using CUDA and are optimized for NVIDIA GPUs. The VisionWorks software provides an ideal tool with which we can evaluate the effectiveness of GPUSync for the real-time scheduling of real-world applications. With support from NVIDIA, we adapted an alpha version of VisionWorks to run atop PGM<sup>RT</sup>, GPUSync, and LITMUS<sup>RT</sup>. In this section, we describe several challenges we faced in this effort, along with our solutions. However, first we wish to note that the alpha version of VisionWorks provided to us by NVIDIA was under active development at the time. *The reader should not assume that statements we make regarding VisionWorks will necessarily hold when the software is made available to the public.* Also, our

---

<sup>11</sup>At line 17 of Figure 5.4(a), the pedestrian detection graph is executed once, from end-to-end, by calling the function `vxProcessGraph()`. This function does not return until the graph has completed. The OpenVX function `vxScheduleGraph()` may be used to *asynchronously* execute a graph without blocking. However, a graph instance must still complete before it may be reissued.

work with VisionWorks was funded by NVIDIA through their internship program—at this time we are unable to share the software we developed in this effort,<sup>12</sup> since it is the property of NVIDIA.

We describe the software we developed to bring real-time support to VisionWorks in three parts. We begin by describing the changes we made to the VisionWorks execution model to support a PGM-derived sporadic task model. Following this, we discuss a separate GPGPU interception library, called *libgpui*, we developed to transparently add GPUSync GPU scheduling to VisionWorks and the third-party libraries it uses. Finally, we describe how our modified VisionWorks execution model directly interacts with GPUSync; specifically, when GPU tokens are acquired and engine locks are obtained.

#### 5.4.1 VisionWorks and the Sporadic Task Model

In this section, we describe the graph execution model used by our alpha version of VisionWorks and how we modified it to support the sporadic task model.

VisionWorks adopts the simple execution model prescribed by the OpenVX specification. Moreover, in the alpha version software, nodes of a graph are executed in topological order by a single thread. However, VisionWorks places no restrictions on the threading model used by individual primitives. We found that several primitives, by way third party libraries such as OpenCV, employ OpenMP to execute for-loops across several parallel threads. In order to remain within the constraints of the single-threaded sporadic task mode, we took the necessary steps to disable such intra-node multi-threading. For example, to control OpenMP, we set the environment variable `OMP_NUM_THREADS = 1`, effectively disabling it.

**Sporadic Task Set Execution Model.** Our first change to VisionWorks was to introduce a new API function, `nvxSpawnGraph()`, which is used to spawn a graph for *periodic* execution. This function is somewhat analogous to the OpenVX asynchronous graph-scheduling function `vxScheduleGraph()`. However, unlike `vxScheduleGraph()`, a graph spawned by `nvxSpawnGraph()` executes repetitively, instead of only once. Each node of a spawned graph is executed by a dedicated thread. Each thread is assigned a common period specified by the programmer. We assume implicit deadlines. An invocation of a node is equivalent to a periodic job (we sometimes use the terms “node” and “job” interchangeably). We call our new execution model for VisionWorks the “sporadic task set execution model.”

---

<sup>12</sup>Specifically, we refer to our modified version of VisionWorks and a software library we call *libgpui*.

**Graph Input and Output.** The OpenVX API assumes that graph input is primed prior to the call of `vxProcessGraph()` (or `vxScheduleGraph()`). For example, in Figure 5.4(a), the contents of the input image `imageRaw` must be set prior to the call to `vxProcessGraph()` on line 17. We achieve a similar behavior by attaching an “input callback” to the spawned graph. The input callback is executed by an “input callback node” that is prepended to the graph—all source nodes of the original graph become children to the input callback node. The same approach is taken to handle graph output, where sink nodes become parents of an appended “output callback node.”

**Graph Dependencies and Pipelining.** We use  $\text{PGM}^{\text{RT}}$  to coordinate the execution of the per-node threads. We achieve this by duplicating the VisionWorks graph structure with an auxiliary graph in  $\text{PGM}^{\text{RT}}$ . We connect the nodes of the  $\text{PGM}^{\text{RT}}$  graph with edges that use event-signaling tokens. We set  $\varrho_i^{k \leftarrow j} = \kappa_i^{k \leftarrow j} = \phi_i^{k \leftarrow j} = 1$  for all edges.

Recall from Section 5.3 that OpenVX does not pass data through graph edges. Rather, node input and output is passed through *singular instances* of data objects. Although graph pipelining is naturally supported by the periodic task set execution model, a new hazard arises: *a producer node may overwrite the contents of a data object before the old contents has been read or written by consumer node!* Such consumers may not even be a direct successor of the producer. For instance, we can conceive of a graph where an image data object is passed through a chain of nodes, each node applying a filter to the image. The node at the head of this chain cannot execute again until after the image has been handled by the node at the tail. In short, the graph cannot be pipelined.

To resolve this pipelining issue, we begin by first *replicating* all data objects in the graph  $N$  times. We set  $N$  to the maximum depth of the graph as a rule of thumb. The number of replicas is actually configurable by the programmer. (Any value for  $N \geq 1$  will work, given the fail-safe mechanism we discuss shortly.) The  $j^{\text{th}}$  invocation of a node (*i.e.*, a job) accesses the  $(j \bmod N)^{\text{th}}$  replica. However, replication alone does not ensure safe pipelining, since we do not enforce end-to-end graph precedence constraints. For example, a node on its  $(j + N)^{\text{th}}$  invocation may execute before the data it generated on its  $j^{\text{th}}$  invocation has been fully consumed. To prevent this from happening, we introduce a fail-safe that stalls the node whenever such hazards arise.

The fail-safe is realized through an additional PGM “feedback graph.” To generate the feedback graph, we first create a *copy* of the auxiliary PGM graph that already shadows the structure of the VisionWorks graph. We then add *additional* edges to this copy. Each additional directed edge connects a node that accesses

```

procedure DOJOB()
  ConsumeTokens();           ▷ Wait for input from producers.
  ConsumeFeedbackTokens();   ▷ Wait for go-ahead from descendants.
  DoPrimitive();             ▷ Execute the primitive.
  ProduceFeedbackTokens();    ▷ Give go-ahead to any waiting ancestors.
  ProduceTokens();           ▷ Signal output to consumers.
   $j \leftarrow j + 1$         ▷ Increment job counter.
end procedure

```

Figure 5.5: Procedure for PGM<sup>RT</sup>-coordinated job execution.

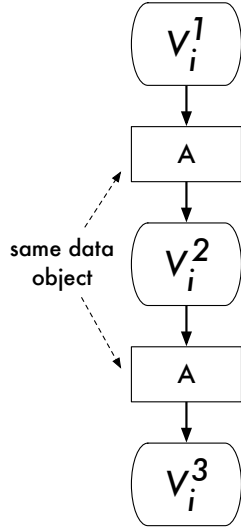
a data object to any *descendant* node that also accesses that data object (no edge is added if such an edge already exists). We then *flip* the direction of all edges in this graph, and each edge is initialized with an available token count equal to  $N$ .

Figure 5.5 outlines the procedure executed by each thread to execute a job. The  $j^{\text{th}}$  invocation of the node first waits for and consumes a token from each of its producers. It then consumes a token from each inbound-edge in the feedback graph, blocking if the requisite tokens are unavailable. The node only blocks if the  $(j \bmod N)^{\text{th}}$  data object replicas are still in use.<sup>13</sup> Thus, the node stalls, and the hazard is avoided. After executing the actual primitive, the node signals that it is done using the  $(j \bmod N)^{\text{th}}$  data object replicas. The node then generates tokens for its consumers.

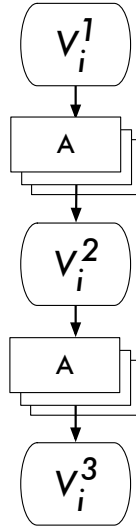
**Example 5.1.** We illustrate the measures we take to support pipelining with an example depicted in Figure 5.6. In Figure 5.6(a), we begin with a simple three-node VisionWorks graph,  $V_i$ . Suppose that node  $V_i^1$  writes to the data object  $A$ ; node  $V_i^2$  modifies  $A$ ; and node  $V_i^3$  reads  $A$ . Graph  $V_i$  has a depth of three. In Figure 5.6(b), we replicate the data object  $A$  such that there are three replicas. Figure 5.6(c) gives the structure of  $V_i$ , without data objects. This structure is replicated in PGM<sup>RT</sup>, represented by graph  $G_i$  in Figure 5.6(d). Token production, consumption, and threshold parameters are set to one. We generate the feedback graph,  $\bar{G}_i$ , as depicted in Figure 5.6(e). Note the edge connecting node  $\bar{G}_i^3$  to  $\bar{G}_i^1$ . ◇

**Support for Back-Edges.** Computer vision algorithms that operate on video streams often feed data derived from prior frames back into the computations performed on future frames. For example, an object tracking algorithm must recall information about objects of prior frames if the algorithm is to describe the motions of those objects in the current frame. OpenVX defines a special data object called a “delay,” which is used

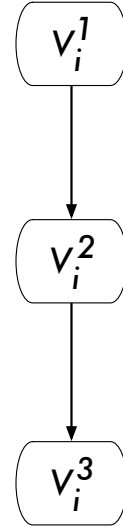
<sup>13</sup>We disable priority boosting and deadline shifting for operations on the feedback graph in order to prevent alterations to properly assigned real-time priorities. The feedback graph is a fail-safe mechanism meant to prevent the overwriting of data before it has been consumed; it is only meant to trigger when task execution behaviors violate provisioned resources (*i.e.*, the number of graph replicas is insufficient). Of course, these special measures only apply to spawned graphs executing under LITMUS<sup>RT</sup>, since these features are specific to LITMUS<sup>RT</sup>.



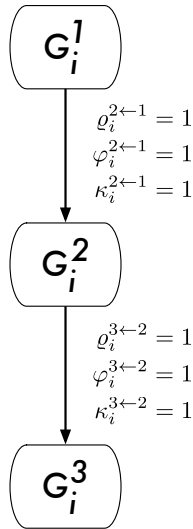
(a) VisionWorks graph,  $V_i$ .



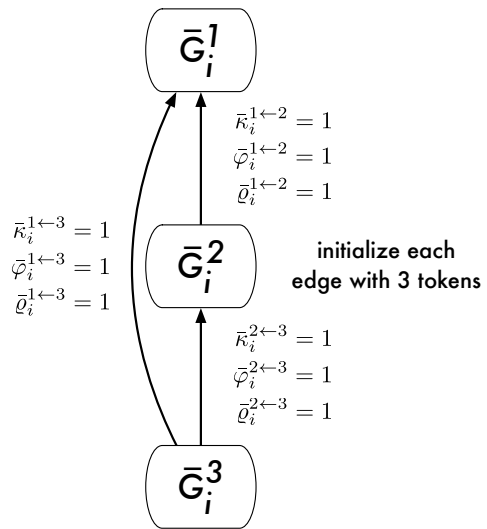
(b)  $V_i$  with replicated data objects.



(c) Structure of  $V_i$  without data objects.



(d) PGM graph,  $G_i$ , of  $V_i$ .



(e) PGM feedback graph,  $\bar{G}_i$ , for  $V_i$ .

Figure 5.6: Derivation of PGM graphs used to support the pipelined thread-per-node execution model.



to buffer node output for use by subsequent node invocations. A delay is essentially a ring buffer used to contain other data objects (*e.g.*, prior image frames). The oldest data object is overwritten when a new data object enters the buffer. The number of data objects stored in a ring buffer (or the “size” of the delay) is tied to how “far into the past” the vision algorithm must go. For example, suppose a node operates on frame  $i$  and it needs to access copies of the last two prior frames. In this case, the size of the delay would be two.

The consumer node of data buffered by a delay may appear anywhere within a graph. It may be an ancestor or descendant of the producer node—it may even be the producer itself. A back-edge is created when the consumer node of a delay is not a descendant of the producer node in the graph derived from non-delay data objects. Such back-edges can be used to implement the object tracking algorithm described above.

Due to complexities in the implementation and use of delays in VisionWorks, for our periodic task set execution model, we do not replicate delay data objects as we do for other types of data objects. Instead, we simply increase the size of the delay ring buffer to store an additional  $N$  data objects. For example, suppose we have a delay with size  $M$ ; we increase the size of the delay to  $M + N$ . We also introduce the necessary back-edges to our auxiliary and feedback PGM graphs. The available token counts for these back-edges must be initialized with the appropriate number of tokens to allow a consuming node to execute while the delay is initially filling with data objects.

**Scheduling Policies.** The programmer may specify which scheduling policy to use for the threads in our sporadic task set execution model. Our modified VisionWorks supports the standard Linux `SCHED_OTHER` and `SCHED_FIFO` policies, as well as `SCHED_LITMUS`. We use POSIX real-time timers to implement periodic job releases under the standard Linux policies. We rely upon the periodic job release infrastructure of `LITMUSRT` for the `SCHED_LITMUS` policy.

We assume deadline-based scheduling under the `SCHED_LITMUS` policy. When the `SCHED_FIFO` policy is employed, the programmer supplies a “base” graph priority. The priority of a thread of a given node is determined by taking the length of the longest path between the node and the input callback source node, plus the base graph priority. Thus, thread priorities increase monotonically down the graph. We use this prioritization scheme to expedite the movement of data down a graph. It is important to finish work deeper in the graph, since graph execution is pipelined.

### 5.4.2 Libgpui: An Interposed GPGPU Library for CUDA

VisionWorks is a large and complex software package. The alpha version of VisionWorks we modified was made up of approximately 80k lines of code, not including third-party libraries. In the evaluation of GPUSync in Chapter 4, the test programs interfaced directly with GPUSync through the user interface provided by liblitmus, as described in Section 3.3.5. We deemed it infeasible to manually alter VisionWorks and GPU-using third-party libraries to use GPUSync directly.

Instead, we developed an interposition library, *libgpui*, to intercept all calls to the lowest-level library of NVIDIA’s CUDA software, *libcuda*. Libgpui intercepts all API calls that may launch a kernel or issue a DMA operation. With the exception of GPU token acquisition, libgpui is entirely transparent to the user—no modifications to user code are necessary. The library automatically interfaces with GPUSync on behalf of the API callee. For example, with features provided by *libcuda*, libgpui inspects the memory address parameters of the DMA API call to deduce the end-points of the DMA operation (*i.e.*, the memories of the source and destination of the DMA) and libgpui automatically requests the needed copy engine locks. Each API call is passed on to *libcuda* once GPUSync schedules the operation. Libgpui also automatically breaks large DMA operations into smaller chunks, as we discussed in Section 3.2.3.4.

Libgpui also overrides three default behaviors of the CUDA runtime to improve real-time predictability. First, libgpui can be configured to enforce suspension-based synchronization when tasks wait for a GPU operation to complete. Second, libgpui ensures that proper stream synchronization behaviors are followed to ensure engine independence (see Section 2.4.4), if it is required for the particular GPU being used. Finally, libgpui forces all GPU operations to be issued on unique per-thread CUDA streams. This is a change from the current CUDA runtime, where threads within a process share a common stream by default.<sup>14</sup> This prevents the threads for contending for access to a shared CUDA stream, which would be arbitrated by the CUDA runtime and result in unpredictable behavior.

### 5.4.3 VisionWorks, libgpui, and GPUSync

We now describe how VisionWorks interacts with GPUSync through libgpui. The alpha version of VisionWorks does not support transparent migration of data objects among GPUs, so we focus our attention

---

<sup>14</sup>Coincidentally, a stream-per-thread feature similar to libgpui’s is slated to be a part of the forthcoming CUDA 7.0 (Harris, 2015).

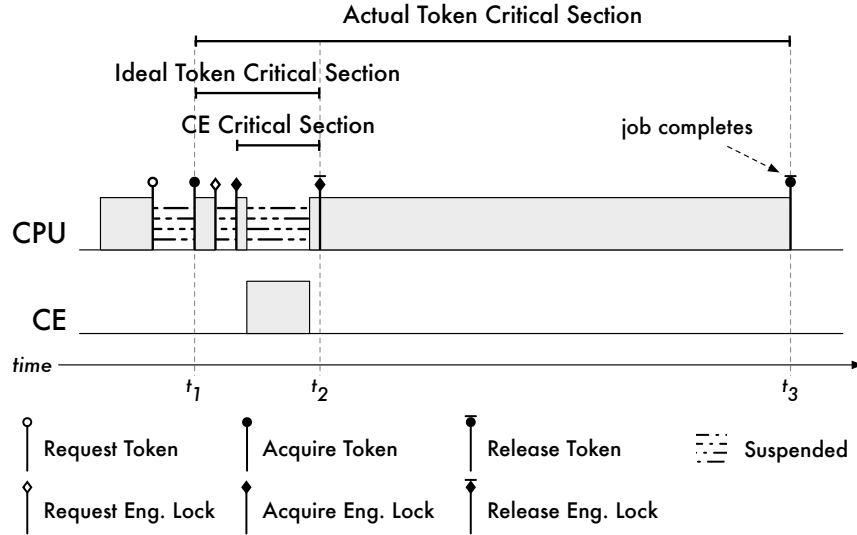


Figure 5.7: Overly long token critical sections may result by releasing tokens at job completion time.

on partitioned GPU scheduling under GPUSync.<sup>15</sup> The user assigns a given VisionWorks application to a GPU partition through an environment variable when the application is launched. Libgpui reads this variable and initializes the CUDA runtime for the selected GPU on behalf of the VisionWorks application.

Although there is only one GPU within each partition, GPUSync still requires each GPU-using job to acquire one of the  $\rho$  tokens before using the GPU. This is necessary to ensure that interrupt and CUDA runtime callback threads are assigned proper real-time priorities. When libgpui intercepts an API call, it first checks to see if the callee already holds a GPU token. If not, libgpui requests and obtains a token before proceeding with the necessary engine lock request. We modified VisionWorks to interface *directly* with libgpui to release any held token upon job completion—this is the only aspect to libgpui that is not transparent to VisionWorks. (Unfortunately, there is no CUDA API libgpui can intercept and interpret as a job completion, so libgpui must be notified explicitly.)

One drawback to using libgpui in this way is that token critical section lengths may be longer than are strictly necessary. We depict such a case in Figure 5.7, where a primitive performs a DMA operation early in its execution and does not use the GPU during the remainder of its computations. We may see such an execution pattern in a primitive that executes primarily on a CPU, but consumes the output of another primitive that executes on a GPU. In Figure 5.7, at time  $t_1$ , the primitive operation executing on the CPU

<sup>15</sup>VisionWorks data-object abstractions do not preclude transparent migration. However, we deemed adding such support to be a non-trivial effort and outside the scope of our core goal of adding real-time GPU scheduling to VisionWorks.

obtains a GPU token. The primitive then copies an OpenVX data object from GPU memory to host memory. This DMA operation completes at time  $t_2$ . The remaining computations of the primitive take place entirely on the CPU, completing at time  $t_3$ . At this point, the GPU token is released. Under our methodology, the GPU token is held during these CPU computations over the time interval  $[t_1, t_3]$ , even though the GPU is no longer required after time  $t_2$ . Ideally, the GPU token would be freed at time  $t_2$ . This is a compromise we make by not integrating GPUSync with VisionWorks directly. We may work around this issue by splitting the primitive into two parts: a DMA primitive and a CPU-computation primitive. However, such a change is invasive—perhaps more invasive than simply modifying the primitive to communicate directly with libgpui to release the token early. Perhaps future versions of VisionWorks or OpenVX could include an API that allows a primitive to express to the execution framework of when a particular heterogeneous computing element (*e.g.*, a GPU) is no longer needed. This API could be leveraged to shorten the token critical section length.

## 5.5 Evaluation of VisionWorks Under GPUSync

In this section, we evaluate the runtime performance of our sporadic task set execution model for VisionWorks under eight configurations of GPUSync. We compare our results against VisionWorks (using the same sporadic task set execution model) running under two purely Linux-based configurations, as well as a LITMUS<sup>RT</sup> configuration *without* GPUSync.

The rest of this section is organized as follows. We begin with a description of the computer vision application we used to evaluate VisionWorks with GPUSync. We then discuss our experimental setup. Finally, we present our results in two parts. In the first part, we report the observed frame-to-frame output delays of many concurrently executed graphs in our experiments. In the second part, we report on observed end-to-end response-time latencies of these same graphs.

### 5.5.1 Video Stabilization

VisionWorks includes a variety of demo applications, including a pedestrian detection application, not unlike the one illustrated in Figure 5.4. However, the pedestrian detection application is relatively uninteresting from a scheduling perspective—it is made up of only a handful of nodes arranged in a pipeline. In our evaluation, we use VisionWorks’ “video stabilization” demo, since the application is far more complex and uses primitives common to other computer vision algorithms. Video stabilization is used to digitally

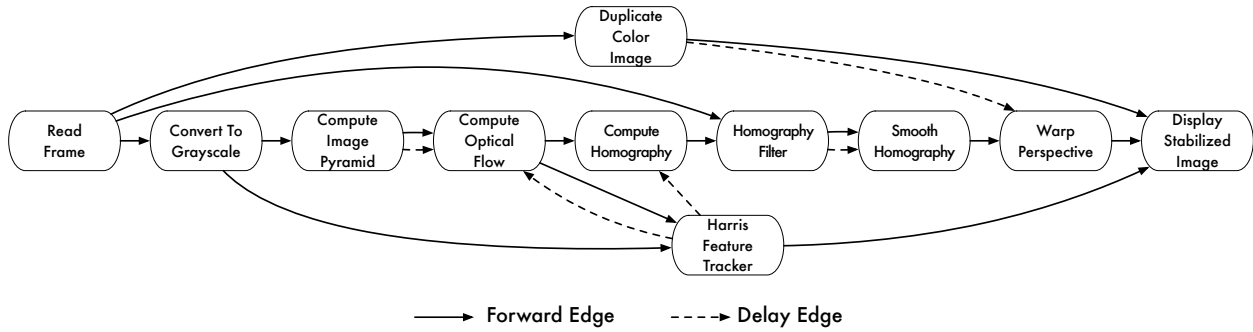


Figure 5.8: Dependency graph of video stabilization application.

dampen the effect of shaky camera movement on a video stream. Vehicle-mounted cameras are prone to camera shake. A video stream may require stabilization as a pre-processing step before higher-level processing is possible. For example, an object tracker may require stabilization—too much shake may decrease the accuracy of predicted object trajectories.

Figure 5.8 depicts the dependency graph of the video stabilization application. Table 5.1 gives a brief description of each node in this graph. We make note of two characteristics of this graph. First, video stabilization operates over a temporal window of several frames. This is needed in order to differentiate between movements due to camera shake and desired camera translation (*i.e.*, stable long-term movement of the camera). These inter-frame dependencies are implemented using OpenVX delay data objects, which are reflected by delay edges in Figure 5.8. Second, although the primitive of a node may execute entirely on CPUs, it may still use a GPU to pull data out of GPU memory through DMA operations. The “Display Stabilized Image” node is such an example. Here, the “Warp Perspective” node performs its computation and stores a stabilized frame in GPU memory. The Display Stabilized Image node pulls this data off of the GPU through DMA. This means that the Display Stabilized Image node will compete with other nodes for GPU tokens, even though it does not use the GPU to perform computation.

### 5.5.2 Experimental Setup

In this evaluation, we focus on clustered CPU scheduling with a single GPU. This focus is motivated by the results of Chapter 4, as well as the inability of our version of VisionWorks to support the migration of data objects among GPUs. Moreover, such a multicore single-GPU system reflects many common embedded computing platforms available today.

Node Name	Function
Read Frame	Reads a frame from the video source.
Duplicate Color Image	Copies the input color image for later use.
Convert To Grayscale	Converts a frame from a color to grayscale ( <i>i.e.</i> , “black and white”) image.
Harris Feature Tracker	Detects Harris corners (features) in an image.
Compute Image Pyramid	Resizes the image into several images at multiple resolutions.
Compute Optical Flow	Determines the movement of image features from the last frame into the current frame.
Compute Homography	Computes a “homography matrix” that characterizes the transformation from the last frame into the current frame.
Homography Filter	Filters noisy values from homography matrix.
Smooth Homography	Merges the homography matrices of the last several frames into one.
Warp Perspective	Transforms an image using a provided homography matrix. (Stabilization occurs here.)
Display Stabilized Image	Displays the stabilized image.

Table 5.1: Description of nodes used in the video stabilization graph of Figure 5.8.

Our experimental workload was comprised of seventeen instances of the video stabilization application in order to load the CPUs and GPU engines. Each graph was executed within its own Linux process. Thus, each instance had eleven regular real-time threads (one for each node in the graph depicted in Figure 5.8) and one CUDA callback thread, which GPUSync automatically schedules as a real-time task (see Section 3.2.5.2). With the inclusion of the GPU interrupt handling thread (see Section 3.2.5.1), there were a total of 205 real-time threads in the workload.

To each graph we assigned a period that was shared by every real-time task of the nodes therein. Two graphs had 20ms periods; another two graphs had 30ms periods; and yet another two graphs had 40ms periods. Five graphs had a period of 60ms; four graphs had a period of 80ms; and another two graphs had a period of 100ms. These periods represent a range of those we find in ADAS, such as those we described in Chapter 1 (see Table 1.1). Table 5.2 summarizes these period assignments.

We isolated the real-time tasks to a single NUMA node of the hardware platform we used in Chapter 4. The remaining NUMA node was used for performance monitoring and did not execute any real-time tasks. The workload was executed across six CPUs and a single NUMA-local GPU. This heavily loaded the CPUs and GPU to near capacity.<sup>16</sup>

The workload was scheduled under eleven different system configurations. Eight of these configurations used GPUSync under LITMUS<sup>RT</sup>’s C-EDF scheduling, with CPU clustering around the L3 cache (*i.e.*, G-EDF

<sup>16</sup>During experimentation, the system tool `top` reported the NUMA node to be 96% utilized (a CPU utilization of 5.76). Similarly, the NVIDIA tool `nvidia-smi` reported the EE to be 66% utilized.

Period	No. Graphs	Base Priority (for SCHED_FIFO only)
20ms	2	75
30ms	2	60
40ms	2	45
60ms	5	30
80ms	4	15
100ms	2	1

Table 5.2: Evaluation task set using VisionWorks’ video stabilization demo.

scheduling within the NUMA node). We experimented with four GPUSync configurations of the GPU Allocator. These configurations differed in the number of tokens,  $\rho$ , and maximum FIFO length,  $f$ . The number of tokens were set to one, two, three, or six. The maximum FIFO length was set correspondingly to six, three, two, or one. We are interested in these GPU Allocator configurations because the associated GPU token blocking terms under each configuration are optimal with respect to suspension-oblivious schedulability analysis for a platform with six CPUs and a single GPU. Under each corresponding pairing of  $\rho$  and  $f$ , we experimented with both FIFO-ordered (FIFO) and priority-ordered (PRIO) engine locks, resulting in the eight GPUSync configurations. We refer to each configuration with a tuple of the form  $(\rho, f, \text{Engine Lock Protocol})$ . For example, a configuration with  $\rho = 2$ ,  $f = 3$ , and priority-ordered engine locks is denoted by  $(2, 3, \text{PRIO})$ . The remaining three system configurations were as follows: LITMUS<sup>RT</sup> C-EDF scheduling without GPUSync; Linux under SCHED\_FIFO fixed-priority scheduling; and Linux under SCHED\_OTHER scheduling. These last three configurations relied upon the GPGPU runtime, GPU driver, and GPU hardware to handle all GPU resource arbitration and scheduling. Under the SCHED\_FIFO policy, graphs were assigned the base priorities given in Table 5.2. A fixed-priority for the thread of each node was derived using the method we described in Section 5.4.1.

We executed our task set under each of the eleven system configurations for 400 seconds. Each graph processed a pre-recorded video file cached in system memory. For all configurations, we used libgpui to force all tasks to suspend while waiting for GPU operations to complete, and to also ensure that all nodes used distinct GPU streams to issue GPU operations. Libgpui invoked GPUSync for only the GPUSync configurations—libgpui passed CUDA API calls immediately on to the underlying CUDA library for non-GPUSync configurations.

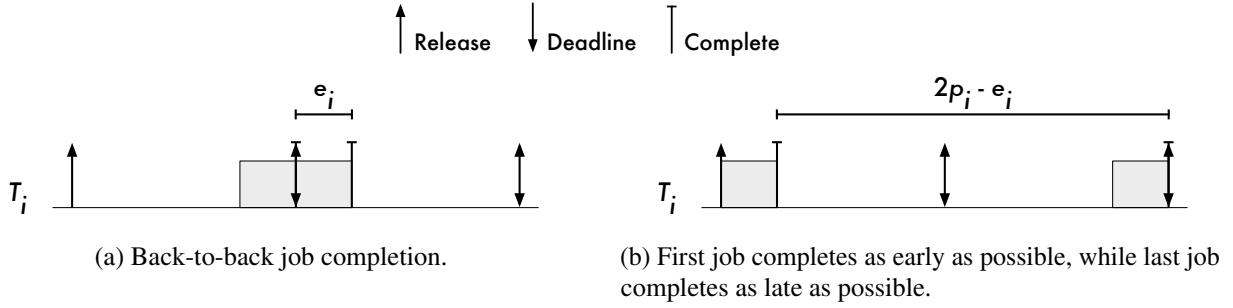


Figure 5.9: Scenarios that lead to extreme values for measured completion delays.

### 5.5.3 Results

We now report our findings in two parts. In the first part, we examine the observed delay between consecutive outputs of the graphs. In the second part, we examine observed end-to-end response time latencies (*i.e.*, the time from the release of a source node to the completion of the corresponding sink node) under the LITMUS<sup>RT</sup>-based configurations.

#### 5.5.3.1 Completion Delays

We require a common observational framework in order to fairly compare the eleven system configurations. Although LITMUS<sup>RT</sup> offers kernel-level low-overhead tracing capabilities, we are unable to make use of them for the non-LITMUS<sup>RT</sup>-based configurations. Instead, we take our measurements from user-space within the graph applications themselves, and we examine the timing properties of the sink node in each graph. Specifically, we look at *the delay between consecutive completions* of the sink node in each graph. To gather these metrics, the sink node of each graph (*i.e.*, the “Display Stabilized Image” node) records a time stamp at the end of its computation. We then measure the difference in consecutive timestamps to obtain a “completion delay” metric.

There are two benefits to this completion delay metric. First, we may make observations from user space; we do not require special support from the operating system, besides access to accurate hardware time counters (which are commonly available). Second, we may apply the completion delay metric to any periodic task set, regardless of any processor schedulers, be they real-time or general purpose. There are two limitations, however. First, completion delay metrics are only useful within the context of periodic task sets, since task periods not only ensure a minimum separation time between job releases of a given task,



but also a *maximum* separation time. We cannot meaningfully compare measured completion delays if the gap between consecutive job releases varies. Second, the metric is inherently noisy. Figure 5.9 depicts two extreme completion delay values that are possible for an implicit-deadline periodic task that is *hard* real-time schedulable. In Figure 5.9(a), the two jobs complete back-to-back. In Figure 5.9(b), the first job completes as early as possible and the second as late as possible. As we see, any completion delay measurement for a job  $J_i$  of a schedulable periodic task set may vary on the interval  $[e_i, 2p_i - e_i]$ . Of course, this interval may grow if the first of two consecutive jobs executes for less than  $e_i$  time units. The upper-bound (*i.e.*,  $2p_i - e_i$ ) may also increase if we consider task sets that are schedulable with bounded tardiness. Despite these limitations, we feel that observed completion delays are useful in reasoning about the real-time runtime performance of our various system configurations.

What makes a “good” completion delay measurement? In these experiments, we desire smooth playback of stabilized video—we want completion delays that are close to the graph period. We also want to see little variation in completion delays. We measure variation by computing the standard deviation ( $\sigma$ ) of measured completion delays of each graph.

Tables 5.3 through 5.13 report the characteristics of the completion delays for each graph under each labeled configuration. The tables include columns for the *maximum*, *99.9<sup>th</sup> percentile*, *99<sup>th</sup> percentile*, *median*, and *mean* observed completion delay for each graph. The tables also include the standard deviation for each mean. The tables also include a column reporting the percentage of video frames that a graph could not complete in the allotted time. For example, in a 400 second experiment, a graph with a period of 20ms should complete 20,000 frames. If it only completes 15,000 frames, then we say that the graph has dropped 5,000 frames, or 25%. A graph with dropped frames is indicative of processor starvation. This gives us another method by which to detect an unschedulable task set observationally. We make the following observations.

**Observation 42.** *The task set is clearly unschedulable under the SCHED\_FIFO, and GPUSync with  $\rho \in \{1, 2\}$  configurations.*

We look at the percentage of dropped frames to detect unschedulability. Each system configuration exhibits different characteristics as to which graphs dropped frames.

In Table 5.4, we see that the graphs with the longest periods (100ms), *i.e.*, those with the lowest fixed priorities, dropped many frames under SCHED\_FIFO. For instance,  $G_{16}$  dropped 88.25% of the 4,000 frames

that should have been processed. This sort of behavior is characteristic of fixed-priority schedulers—the lowest priority tasks may be starved of CPU time.

In Tables 5.6 and 5.7, we see that *many* frames are dropped by all tasks under GPUSync with  $\rho = 1$ .<sup>17</sup> The percentage of dropped frames increases with period. For example,  $G_1$  in Table 5.6 drops 25.41% of its frames, while  $G_{17}$  drops about 47.33%. Although performance is generally bad under these configurations, we do observe that the number of dropped frames is strongly correlated with graph period. That is, performance is very regular. For instance, the five graphs in Table 5.7 with a 60ms period (graphs  $G_7$  through  $G_{11}$ ) each drop either 44.76% or 44.78% of frames—they all exhibit nearly the same behavior. Similar regularity is reflected by the median and mean completion delays for these tasks.

In Tables 5.8 and 5.9, we see that performance improves with a greater number of GPU tokens, where  $\rho = 2$ . However, frames are still dropped. We observe a similar regularity in the percentage of dropped frames as we saw in Tables 5.6 and 5.7 (GPUSync with  $\rho = 1$ ).

**Observation 43.** *GPUSync, with  $\rho \in \{3, 6\}$ , improved observed predictability.*

We can observe this by looking at several different metrics. We first compare GPUSync with  $\rho \in \{3, 6\}$  (Tables 5.10 through 5.13) against LITMUS<sup>RT</sup> without GPUSync (Table 5.5). These system configurations all share the same CPU scheduler (C-EDF), yet there are clear differences in runtime behavior.

We observe significant differences in terms of outlier behavior. For example, in Table 5.5 for LITMUS<sup>RT</sup> without GPUSync,  $G_7$  has a maximum, 99.9<sup>th</sup> percentile, and 99<sup>th</sup> percentile completion delays of 1108.68ms, 776.63ms, and 465.71ms, respectively. In Table 5.13 for GPUSync with (6, 1, PRIO),  $G_7$  has a maximum, 99.9<sup>th</sup> percentile, and 99<sup>th</sup> percentile completion delays of 224.33ms, 192.2ms, 88.0ms, respectively. The observed completion delays for  $G_7$  under GPUSync with (6, 1, PRIO) were reduced by factors of approximately 4.9, 4.0, and 5.3 for maximum, 99.9<sup>th</sup> percentile, and 99<sup>th</sup> percentile measurements, respectively. We see similar improvements when we consider GPUSync with (3, 2, FIFO), (3, 2, PRIO), and (6, 1, FIFO).

We also compare the behavior of GPUSync with  $\rho \in \{3, 6\}$  against LITMUS<sup>RT</sup> without GPUSync in terms of the standard deviation of completion delays. In Table 5.5 for LITMUS<sup>RT</sup> without GPUSync, the standard deviations range over [15.47ms, 130.21ms]. In Table 5.10 for GPUSync with (3, 2, FIFO), the range

---

<sup>17</sup>Since  $\rho = 1$ , these configurations are theoretically equivalent since a token-holding job immediately receives every engine lock it requests. That is, the token grants exclusive access to the GPU. However, we do see some minor variation. These variations may be simply due to experimental noise or differences in runtime overheads due to different locking logic behind FIFO and PRIO engine locks.

is [6.21ms, 10.07ms]. In Table 5.11 for GPUSync with (3,2,PRIO), the range is [6.32ms, 10.55ms]. In Table 5.12 for GPUSync with (6,1,FIFO), the range is [8.11ms, 14.65ms]. In Table 5.13 for GPUSync with (6,1,PRIO), the range is [10.64ms, 16.94ms]. With the exception of GPUSync with (6,1,PRIO), the *greatest* standard deviations of these GPUSync configurations are less than the *smallest* standard deviation under LITMUS<sup>RT</sup> without GPUSync.

From these observations, we conclude that *real-time CPU scheduling alone is not enough to ensure real-time perceptibility in a system with GPUs*.

We make similar comparisons of GPUSync with  $\rho \in \{3,6\}$  against SCHED\_OTHER (Table 5.3), and we also observe significant differences in terms of outlier behavior. For example, in Table 5.3 for SCHED\_OTHER,  $G_1$  has a maximum, 99.9<sup>th</sup> percentile, and 99<sup>th</sup> percentile completion delays of 405.88ms, 233.32ms, and 54.61ms, respectively. In Table 5.10 for GPUSync with (3,2,FIFO),  $G_1$  has a maximum, 99.9<sup>th</sup> percentile, and 99<sup>th</sup> percentile completion delays of 115.11ms, 83.21ms, 33.81ms, respectively. GPUSync with (3,2,FIFO) reduces maximum, 99.9<sup>th</sup> percentile, and 99<sup>th</sup> percentile completion delays by factors of approximately 3.5, 2.8, and 1.6 times, respectively. We see similar improvements when we consider GPUSync with (3,2,PRIO), (6,1,FIFO), and (6,1,PRIO).

We may also compare the behavior of GPUSync with  $\rho \in \{3,6\}$  against SCHED\_OTHER in terms of the standard deviation of completion delays. In Table 5.3 for SCHED\_OTHER, the standard deviations range over [13.90ms, 18.36ms]. In Table 5.10 for GPUSync with (3,2,FIFO), the range is [6.21ms, 10.07ms]. In Table 5.11 for GPUSync with (3,2,PRIO), the range is [6.32ms, 10.55ms]. In Table 5.12 for GPUSync with (6,1,FIFO), the range is [8.11ms, 14.65ms]. In Table 5.13 for GPUSync with (6,1,PRIO), the range is [10.64ms, 16.94ms]. Although the standard deviation ranges among these GPUSync configurations differ, they are less than that of SCHED\_OTHER. Indeed, the greatest standard deviations when  $\rho = 3$  are less than the smallest standard deviation under SCHED\_OTHER.

**Observation 44.** *In this experiment, the SCHED\_OTHER configuration outperformed both real-time configurations that lack real-time GPU management.*

A surprising result from these experiments is that the SCHED\_OTHER configuration (Table 5.3) outperforms both SCHED\_FIFO (Table 5.4) and LITMUS<sup>RT</sup> without GPUSync (Table 5.5) configurations. The SCHED\_FIFO configuration dropped frames while the SCHED\_OTHER configuration did not. The LITMUS<sup>RT</sup> without GPUSync configuration has worse outlier behavior than the SCHED\_OTHER configuration.

These differences in behavior may be due to busy-waiting employed by the CUDA runtime or Vision-Works software, despite the fact that libgpui forces tasks to suspend while waiting for GPU operations to complete. Under SCHED\_FIFO and LITMUS<sup>RT</sup> without GPUSync configurations, the CPU scheduler always schedules the  $m$ -highest priority tasks (in this experiment,  $m = 6$ ) that are ready to run. CPU time may be wasted if any of these tasks busy-wait for a long duration of time. The amount of CPU time wasted by a task at the expense of other tasks is limited under SCHED\_OTHER, since the scheduler attempts to distribute CPU time equally among all tasks. That is, the scheduler will preempt a task that busy-waits for too long. This assumes that the tasks use *preemptive* busy-waiting, but this would be expected of software developed for general purpose computing.<sup>18</sup> Interestingly, the fact that the SCHED\_FIFO and LITMUS<sup>RT</sup> without GPUSync configurations do not deadlock suggests that any such busy-waiting is probably not used to implement a locking protocol. (Preemptive busy-waiting in a locking protocol under real-time scheduling can easily lead to deadlock.)

It is difficult to draw general conclusions from the completion delay data presented in Tables 5.3 through 5.13 because there is so much data. To help us gain additional insights into the performance of the eleven system configurations, we collapsed the information in the above eleven tables into a single table, Table 5.14. We collapse the data with the following process. We compute the total percentage of dropped

<sup>18</sup>Non-preemptive execution generally requires privileged permissions (e.g., “superuser” permissions).

Graph	Period	% Dropped	Max	99.9 <sup>th</sup> %	99 <sup>th</sup> %	Median	Mean	$\sigma$
$G_1$	20	0	405.88	233.32	54.61	15.94	20.00	15.85
$G_2$	20	0	412.24	248.62	53.06	16.12	20.00	15.57
$G_3$	30	0	377.97	246.86	62.68	28.34	29.99	17.33
$G_4$	30	0	399.43	240.92	64.64	28.08	30.00	17.71
$G_5$	40	0	397.06	291.17	71.02	40.03	40.00	17.75
$G_6$	40	0	452.15	295.14	69.62	39.94	39.99	18.13
$G_7$	60	0	473.23	376.42	82.87	60.07	59.99	17.68
$G_8$	60	0	453.20	367.44	85.32	60.47	59.99	17.91
$G_9$	60	0	414.78	352.29	86.16	60.36	59.99	17.14
$G_{10}$	60	0	462.29	372.52	84.26	60.50	59.99	17.91
$G_{11}$	60	0	454.22	331.98	87.79	59.99	59.99	17.66
$G_{12}$	80	0	485.56	338.03	107.36	79.86	79.98	17.68
$G_{13}$	80	0	442.45	340.89	105.01	79.87	79.98	16.90
$G_{14}$	80	0	391.75	349.92	105.96	79.78	79.98	16.48
$G_{15}$	80	0	486.90	351.47	106.70	79.62	80.00	18.36
$G_{16}$	100	0	440.83	256.71	127.19	100.00	99.99	15.48
$G_{17}$	100	0	376.85	273.21	127.43	99.92	99.98	13.90

Table 5.3: Completion delay data for SCHED\_OTHER. Time in ms.

Graph	Period	% Dropped	Max	99.9 <sup>th</sup> %	99 <sup>th</sup> %	Median	Mean	$\sigma$
$G_1$	20	0	99.25	96.09	25.68	20.26	20.00	4.78
$G_2$	20	0	99.29	95.93	25.86	20.34	20.00	4.88
$G_3$	30	0	115.89	105.92	38.39	28.93	30.00	6.60
$G_4$	30	0	116.27	105.41	38.72	28.90	30.00	6.81
$G_5$	40	0	184.15	140.43	45.91	38.92	40.00	6.73
$G_6$	40	0	184.30	142.32	45.89	38.94	40.00	6.69
$G_7$	60	0	533.52	387.16	96.62	59.22	59.98	21.72
$G_8$	60	0	456.71	351.90	96.87	59.58	60.00	20.61
$G_9$	60	0	519.93	347.87	94.97	59.24	59.99	20.36
$G_{10}$	60	0	463.48	346.12	91.02	59.76	59.99	20.61
$G_{11}$	60	0	396.98	353.92	96.65	59.63	59.99	19.86
$G_{12}$	80	0	904.76	770.43	204.38	66.19	79.98	49.61
$G_{13}$	80	0	964.77	729.72	213.20	66.37	79.95	49.63
$G_{14}$	80	0	903.07	771.49	202.04	66.64	79.93	48.67
$G_{15}$	80	0	936.06	717.49	233.15	66.59	79.94	49.44
$G_{16}$	100	88.25	3786.02	3786.02	2279.03	920.63	850.65	559.04
$G_{17}$	100	88.25	3844.70	3844.70	2333.78	879.83	848.36	580.58

Table 5.4: Completion delay data for SCHED\_FIFO. Time in ms.

Graph	Period	% Dropped	Max	99.9 <sup>th</sup> %	99 <sup>th</sup> %	Median	Mean	$\sigma$
$G_1$	20	0	612.16	300.41	159.49	4.62	20.00	34.69
$G_2$	20	0	254.20	143.12	62.49	19.30	20.00	15.47
$G_3$	30	0	309.02	242.21	91.72	27.21	30.00	22.61
$G_4$	30	0	272.94	170.40	73.96	27.71	30.00	19.36
$G_5$	40	0	450.22	295.02	135.39	37.88	40.00	31.04
$G_6$	40	0	497.19	335.93	154.33	36.91	40.00	38.84
$G_7$	60	0	1108.68	776.63	465.71	43.80	59.96	85.33
$G_8$	60	0	785.02	596.92	256.90	56.42	59.96	67.26
$G_9$	60	0	459.19	373.74	152.66	58.97	59.95	33.11
$G_{10}$	60	0	383.78	307.72	143.46	58.83	59.95	25.82
$G_{11}$	60	0	383.59	307.70	143.25	58.81	59.95	25.81
$G_{12}$	80	0	703.36	543.54	257.50	70.84	79.86	65.42
$G_{13}$	80	0	861.65	652.05	315.17	69.61	79.86	76.18
$G_{14}$	80	0	829.90	601.26	264.91	70.27	79.86	70.62
$G_{15}$	80	0	865.79	666.63	363.45	69.41	79.86	79.94
$G_{16}$	100	0	1278.93	1077.55	641.47	79.46	99.78	124.68
$G_{17}$	100	0	1416.51	1014.76	711.54	79.27	99.78	130.21

Table 5.5: Completion delay data for LITMUS<sup>RT</sup> without GPUSync. Time in ms.

Graph	Period	% Dropped	Max	99.9th%	99th%	Median	Mean	$\sigma$
$G_1$	20	25.41	413.27	200.79	43.21	25.48	26.90	14.83
$G_2$	20	25.41	406.52	200.29	43.57	25.47	26.90	15.09
$G_3$	30	36.42	422.43	345.47	113.72	45.70	47.35	18.18
$G_4$	30	36.42	422.15	270.20	117.96	45.75	47.34	17.43
$G_5$	40	41.23	435.51	364.95	200.15	66.22	68.29	20.48
$G_6$	40	41.24	436.16	356.91	204.60	66.11	68.29	20.16
$G_7$	60	44.82	462.04	454.50	238.66	105.89	109.09	24.54
$G_8$	60	44.48	460.74	453.70	238.53	105.84	108.58	25.26
$G_9$	60	44.48	458.31	450.14	238.97	106.00	108.58	25.21
$G_{10}$	60	44.48	451.62	442.64	238.14	105.95	108.58	25.28
$G_{11}$	60	44.49	455.47	448.70	241.87	105.83	108.60	25.67
$G_{12}$	80	46.16	492.02	479.68	285.07	145.90	149.23	29.64
$G_{13}$	80	46.16	491.48	479.88	281.86	145.69	149.23	29.51
$G_{14}$	80	46.16	492.47	473.16	281.49	145.84	149.22	28.73
$G_{15}$	80	46.16	471.79	462.25	285.98	145.79	149.22	29.47
$G_{16}$	100	47.33	519.71	498.72	329.67	186.90	190.60	33.30
$G_{17}$	100	47.33	543.10	508.17	324.20	186.96	190.59	33.38

Table 5.6: Completion delay data for GPUSync for (1, 6, FIFO). Time in ms.

Graph	Period	% Dropped	Max	99.9 <sup>th</sup> %	99 <sup>th</sup> %	Median	Mean	$\sigma$
$G_1$	20	25.3	407.33	200.10	43.65	25.22	26.87	14.89
$G_2$	20	25.3	357.44	199.38	43.77	25.79	26.87	14.81
$G_3$	30	36.35	418.08	276.83	119.78	45.81	47.32	17.38
$G_4$	30	36.35	420.18	282.08	113.06	45.77	47.32	18.07
$G_5$	40	41.18	429.04	356.54	203.34	66.12	68.27	20.12
$G_6$	40	41.17	436.99	352.97	201.51	66.29	68.26	20.07
$G_7$	60	44.76	462.70	452.84	237.07	105.94	109.00	24.34
$G_8$	60	44.76	458.72	454.74	239.81	105.86	109.00	24.54
$G_9$	60	44.76	451.96	450.42	239.74	105.90	109.00	24.48
$G_{10}$	60	44.78	452.22	444.57	237.70	105.96	109.03	24.37
$G_{11}$	60	44.78	455.94	449.44	239.17	105.81	109.03	24.68
$G_{12}$	80	46.44	485.87	473.34	285.59	145.74	149.81	27.96
$G_{13}$	80	46.44	495.53	479.42	279.90	145.98	149.81	27.58
$G_{14}$	80	46.44	499.83	478.61	282.22	145.82	149.81	28.22
$G_{15}$	80	46.44	544.97	474.02	282.61	145.84	149.81	28.30
$G_{16}$	100	47.63	522.80	515.44	328.67	187.30	191.44	31.31
$G_{17}$	100	47.63	566.30	515.94	327.71	187.12	191.44	31.92

Table 5.7: Completion delay data for GPUSync for (1, 6, PRIO). Time in ms.

Graph	Period	% Dropped	Max	99.9 <sup>th</sup> %	99 <sup>th</sup> %	Median	Mean	$\sigma$
$G_1$	20	0.99	174.27	93.66	30.40	19.84	20.29	6.00
$G_2$	20	1.06	124.28	96.09	30.85	19.86	20.31	6.09
$G_3$	30	1.14	180.10	111.63	43.56	30.00	30.49	7.09
$G_4$	30	1.15	178.34	108.24	44.06	29.96	30.49	7.00
$G_5$	40	1.37	183.82	120.26	66.28	40.05	40.76	7.64
$G_6$	40	1.33	206.13	122.24	61.22	40.05	40.74	7.62
$G_7$	60	7.32	218.13	170.21	109.48	63.10	65.03	10.55
$G_8$	60	7.28	215.98	162.14	105.90	63.04	65.00	10.26
$G_9$	60	7.35	216.59	162.00	112.48	63.11	65.04	10.54
$G_{10}$	60	7.29	218.09	152.88	109.10	63.02	65.00	10.32
$G_{11}$	60	7.31	217.27	167.35	109.57	63.11	65.01	10.35
$G_{12}$	80	22.66	248.40	217.49	162.55	102.92	103.85	13.80
$G_{13}$	80	22.64	250.84	208.36	156.96	103.03	103.82	13.55
$G_{14}$	80	22.58	250.55	208.46	155.46	102.80	103.72	13.53
$G_{15}$	80	22.64	246.19	223.89	158.22	102.86	103.80	14.05
$G_{16}$	100	30.08	275.23	240.95	208.00	142.62	143.53	15.73
$G_{17}$	100	30.05	244.98	229.37	200.21	142.66	143.46	15.31

Table 5.8: Completion delay data for GPUSync for (2, 3, FIFO). Time in ms.

Graph	Period	% Dropped	Max	99.9 <sup>th</sup> %	99 <sup>th</sup> %	Median	Mean	$\sigma$
$G_1$	20	1.12	188.25	102.50	30.83	19.88	20.32	6.36
$G_2$	20	1.07	178.11	105.39	30.85	19.85	20.31	6.40
$G_3$	30	1.13	197.98	111.49	43.57	29.97	30.49	7.23
$G_4$	30	1.19	162.21	115.02	44.53	29.98	30.51	7.30
$G_5$	40	1.42	191.41	133.79	63.56	39.98	40.78	8.30
$G_6$	40	1.47	214.40	126.83	65.99	40.02	40.80	8.09
$G_7$	60	7.89	216.31	193.13	108.98	63.35	65.43	10.87
$G_8$	60	7.89	218.85	190.42	105.69	63.41	65.43	10.75
$G_9$	60	7.86	222.21	186.86	111.05	63.51	65.41	10.70
$G_{10}$	60	7.95	214.20	170.20	114.40	63.54	65.47	10.78
$G_{11}$	60	7.89	226.12	197.13	108.45	63.47	65.43	10.79
$G_{12}$	80	23.08	255.15	222.58	157.23	103.61	104.41	13.77
$G_{13}$	80	23.08	238.12	228.92	154.70	103.48	104.41	13.98
$G_{14}$	80	23.06	248.43	229.53	154.51	103.64	104.39	13.72
$G_{15}$	80	23.04	261.14	238.38	154.46	103.40	104.35	14.16
$G_{16}$	100	30.25	308.29	261.81	204.78	142.94	143.88	15.95
$G_{17}$	100	30.30	275.49	256.45	210.65	142.91	143.98	16.34

Table 5.9: Completion delay data for GPUSync for (2, 3, PRIO). Time in ms.

Graph	Period	% Dropped	Max	99.9 <sup>th</sup> %	99 <sup>th</sup> %	Median	Mean	$\sigma$
$G_1$	20	0	115.11	83.21	33.81	19.56	20.01	6.21
$G_2$	20	0	154.24	89.77	32.45	19.83	20.01	6.41
$G_3$	30	0	119.40	91.59	44.81	29.55	30.01	6.26
$G_4$	30	0	119.96	90.32	45.51	29.60	30.01	6.48
$G_5$	40	0	154.99	93.90	56.19	39.60	40.02	6.30
$G_6$	40	0	176.54	95.40	55.83	39.70	40.01	6.54
$G_7$	60	0	299.21	118.69	78.54	59.51	60.04	7.09
$G_8$	60	0	209.89	125.10	79.49	59.68	60.04	7.10
$G_9$	60	0	246.73	112.11	79.03	59.57	60.03	6.98
$G_{10}$	60	0	175.79	130.78	80.90	59.72	60.04	7.34
$G_{11}$	60	0	205.56	122.91	79.30	59.60	60.03	6.66
$G_{12}$	80	0	152.52	141.36	105.04	79.44	80.01	8.19
$G_{13}$	80	0	198.76	141.15	104.87	79.63	80.01	7.93
$G_{14}$	80	0	176.11	142.24	106.64	79.55	80.01	8.42
$G_{15}$	80	0	226.29	152.43	103.39	79.55	80.02	8.52
$G_{16}$	100	0	209.92	172.15	129.92	99.57	100.07	10.07
$G_{17}$	100	0	209.91	170.36	131.09	99.59	100.07	9.85

Table 5.10: Completion delay data for GPUSync for (3, 2, FIFO). Time in ms.

Graph	Period	% Dropped	Max	99.9 <sup>th</sup> %	99 <sup>th</sup> %	Median	Mean	$\sigma$
$G_1$	20	0	100.51	86.71	33.42	19.09	20.01	6.32
$G_2$	20	0	171.95	82.87	34.49	19.37	20.01	6.50
$G_3$	30	0	163.08	91.38	44.38	29.25	30.01	6.84
$G_4$	30	0	119.83	95.17	44.77	29.62	30.01	6.62
$G_5$	40	0	216.41	101.41	55.15	40.46	40.02	7.35
$G_6$	40	0	238.03	95.13	56.15	39.73	40.02	7.60
$G_7$	60	0	215.71	120.58	81.55	59.57	60.00	6.70
$G_8$	60	0	295.60	122.66	80.61	59.61	60.01	7.99
$G_9$	60	0	210.39	121.58	81.13	59.50	60.01	7.37
$G_{10}$	60	0	182.70	122.54	79.40	59.59	59.99	6.21
$G_{11}$	60	0	142.32	119.76	78.38	59.58	59.98	6.08
$G_{12}$	80	0	175.73	146.69	106.27	79.47	80.00	8.65
$G_{13}$	80	0	163.97	148.16	109.05	79.78	80.00	8.94
$G_{14}$	80	0	170.10	147.12	109.53	79.56	80.00	9.93
$G_{15}$	80	0	218.05	152.67	109.07	79.62	80.01	9.59
$G_{16}$	100	0	208.42	168.30	139.57	99.89	100.07	10.55
$G_{17}$	100	0	201.03	163.87	135.47	99.80	100.06	10.13

Table 5.11: Completion delay data for GPUSync for (3, 2, PRIO). Time in ms.



Graph	Period	% Dropped	Max	99.9 <sup>th</sup> %	99 <sup>th</sup> %	Median	Mean	$\sigma$
$G_1$	20	0	84.66	69.29	39.55	18.56	20.01	8.11
$G_2$	20	0	100.91	71.41	40.40	18.31	20.01	8.28
$G_3$	30	0	106.65	78.04	49.33	29.45	30.02	9.32
$G_4$	30	0	117.70	67.74	49.16	29.97	30.02	9.33
$G_5$	40	0	188.84	87.52	58.98	40.21	40.02	9.21
$G_6$	40	0	242.19	80.76	58.93	40.09	40.03	9.14
$G_7$	60	0	378.94	120.78	83.18	60.86	60.03	9.42
$G_8$	60	0	378.56	122.67	82.59	60.77	60.03	8.89
$G_9$	60	0	378.50	120.32	84.30	60.93	60.03	10.73
$G_{10}$	60	0	304.69	123.79	85.33	60.45	60.02	9.47
$G_{11}$	60	0	163.24	126.17	89.90	60.42	59.99	10.48
$G_{12}$	80	0	166.87	135.46	115.25	78.99	79.95	11.93
$G_{13}$	80	0	189.56	151.96	115.58	78.77	79.95	12.50
$G_{14}$	80	0	173.29	146.28	112.74	78.33	79.95	12.25
$G_{15}$	80	0	170.68	148.38	105.88	78.12	79.95	10.10
$G_{16}$	100	0	242.56	197.47	139.18	100.00	99.98	14.51
$G_{17}$	100	0	240.90	209.31	140.24	100.09	100.00	14.65

Table 5.12: Completion delay data for GPUSync for (6, 1, FIFO). Time in ms.

Graph	Period	% Dropped	Max	99.9 <sup>th</sup> %	99 <sup>th</sup> %	Median	Mean	$\sigma$
$G_1$	20	0	126.79	113.19	44.66	16.83	20.01	10.64
$G_2$	20	0	136.47	118.97	45.99	17.26	20.01	11.56
$G_3$	30	0	156.19	120.81	52.68	29.89	30.01	11.72
$G_4$	30	0	141.65	113.71	53.14	30.12	30.01	11.68
$G_5$	40	0	174.46	102.46	60.99	41.35	40.01	11.76
$G_6$	40	0	177.19	135.44	61.72	41.84	40.02	12.68
$G_7$	60	0	224.33	192.20	88.00	60.66	59.99	11.08
$G_8$	60	0	224.36	184.27	88.33	60.66	59.99	11.09
$G_9$	60	0	197.53	164.71	88.50	60.72	59.99	10.96
$G_{10}$	60	0	215.30	169.92	89.38	60.64	59.99	11.16
$G_{11}$	60	0	214.23	184.25	89.54	60.71	59.99	11.56
$G_{12}$	80	0	237.41	199.40	116.73	77.81	79.96	15.12
$G_{13}$	80	0	216.04	178.55	116.64	77.79	79.93	13.55
$G_{14}$	80	0	224.25	181.11	116.91	77.82	79.93	13.98
$G_{15}$	80	0	237.55	214.03	116.46	77.25	79.93	14.11
$G_{16}$	100	0	286.73	267.92	147.47	100.18	99.98	16.94
$G_{17}$	100	0	257.77	234.21	139.80	100.26	99.97	16.24

Table 5.13: Completion delay data for GPUSync for (6, 1, PRIO). Time in ms.

Configuration	Total % Dropped	Normalized and Averaged					
		Max	99.9 <sup>th</sup> %	99 <sup>th</sup> %	Median	Mean	$\sigma$
SCHED_OTHER	0	9.20	6.33	1.65	0.97	<b>1.00</b>	0.37
SCHED_FIFO	4.77	11.12	9.83	4.25	1.89	1.88	0.99
LITMUS <sup>RT</sup> C-EDF	0	12.12	8.40	4.22	0.86	<b>1.00</b>	0.93
<i>GPUSync</i>							
(1, 6, FIFO)	37.83	9.65	7.64	3.70	1.69	1.74	0.47
(1, 6, PRIO)	37.88	9.53	7.53	3.70	1.69	1.74	0.46
(2, 3, FIFO)	6.99	4.26	3.05	1.78	1.13	1.15	0.20
(2, 3, PRIO)	7.23	4.54	3.34	1.78	1.14	1.16	0.20
(3, 2, FIFO)	0	<b>3.67</b>	2.37	<b>1.39</b>	<b>0.99</b>	<b>1.00</b>	<b>0.15</b>
(3, 2, PRIO)	0	3.83	2.38	1.41	<b>0.99</b>	<b>1.00</b>	0.16
(6, 1, FIFO)	0	3.99	<b>2.22</b>	1.51	<b>0.99</b>	<b>1.00</b>	0.21
(6, 1, PRIO)	0	3.92	3.23	1.60	0.98	<b>1.00</b>	0.26

Table 5.14: Average normalized completion delay data.

frames by summing the number of dropped frames of all graphs under a given system configuration and computing its share of the total number of frames that should have been completed within the allotted time. To combine the completion delay data, we first *normalized* each measurement by dividing the measurements by the graph period. We then compute the average of the normalized values. (We use the term “average” to differentiate from the completion delay means that we discuss.) For example, the “Max” for SCHED\_OTHER in Table 5.14 reflects the average normalized-maximum of the “Max” values in Table 5.3. Likewise, the standard deviation of GPUSync with (1, 6, PRIO) in Table 5.14 reflects an average of normalized standard deviations from Table 5.13. In Table 5.14, we also highlight the “best” values in each column. We consider values closest to 1.0 as best for average normalized completion delays, and values closest to zero as best for standard deviations. We make the following additional observations.

**Observation 45.** *For GPUSync configurations, completion delays are more regular when  $\rho = 3$ .*

In Table 5.14, we observe that the average normalized maximum completion delay is smallest under GPUSync with (3, 2, FIFO). The next smallest is the GPUSync (3, 2, PRIO) configuration. The same holds for the average normalized 99<sup>th</sup> percentile completion delay and standard deviation. We note that the GPUSync (6, 1, FIFO) configuration exhibited the best average normalized 99.9<sup>th</sup> percentile completion delay. However, given that this configuration is beaten or matched by GPUSync configurations with  $\rho = 3$  in other measures, we assert that  $\rho = 3$  still gives the best real-time performance overall.

It appears that  $\rho = 3$  is the “sweet spot” for  $\rho$  in this experiment. The experimental workload is unschedulable when  $\rho = 2$ , yet real-time performance worsens when  $\rho = 6$ . The likelihood that GPU

resources are left idle is greater with smaller values of  $\rho$ . When  $\rho = 1$ , two of the GPU’s three engines (one EE and two CEs) are guaranteed to be left idle. Similarly, at least one engine will always be idle when  $\rho = 2$ . Engines may still be left idle when  $\rho = 3$ , but the *possibility* remains for all engines to be used simultaneously. However, this line of reasoning fails when we consider the case when  $\rho = 6$ . In Section 3.2.3.2, we argued that we should constrain the number of GPU tokens in order to limit the aggressiveness of migrations. This is no longer a concern in a single-GPU system, so why does performance not continue to improve when  $\rho = 6$ ? We provide the following possible explanation.

In this experiment, we executed the *same amount of work* under GPUSync configurations with  $\rho = 3$  and  $\rho = 6$ . More tokens allows a finite amount work (on both CPUs and GPU engines) to be *shifted* to an earlier point in time of the schedule, since the CPU scheduler and locking protocols are work-conserving. Here, the shifted work “piles up” at an earlier point in the schedule. From Table 5.14, we see that  $\rho = 3$  provides sufficient GPU parallelism to achieve good real-time performance; the constrained number of tokens meters out GPU access. However, when  $\rho = 6$ , bursts in GPU activity, where all GPU engines are used simultaneously, become more likely. These bursts may result in moments of heavy interference (particularly on the system memory bus) with other tasks, thus increasing *variance* in completion delays. This interference would only increase the likelihood of outlier behavior on both ends of the spectrum of observed completion delays, since bursts in GPU activity would be followed by corresponding lulls. We theorize that median-case performance would remain unaffected. This explanation is borne out by the data in Table 5.14, where we see that the average normalized median completion delays under  $\rho = 6$  are generally indistinguishable from configurations where  $\rho = 3$ .

**Observation 46.** *For GPUSync configurations, FIFO engine locks offered better observed predictability than PRIO engine locks.*

To see this, we compare the GPUSync configurations that only differ by engine lock protocol. With exception of configurations where  $\rho = 1$  (which we discuss shortly), we see that the average normalized standard deviations of the mean completion delays are less (or equal in the case of  $\rho = 2$ ) under FIFO. FIFO-ordered engine locks impart very regular blocking behaviors—any single engine request may only be delayed between zero and  $\rho - 1$  other engine requests. Under priority-ordered locks, higher-priority requests may continually “cut ahead” of a low-priority engine request. Thus, the low-priority engine request may be

delayed by more than  $\rho - 1$  requests. A job with such requests may still meet its deadline, but there may be increased variance in completion delays due to increased variance in blocking delays.

**Observation 47.** *GPUSync with  $(3, 2, \text{FIFO})$  outperformed the other ten system configurations.*

In Table 5.14, we see that the GPUSync configuration with  $(3, 2, \text{FIFO})$  produced the smallest Max, 99<sup>th</sup>, and standard deviation values among all configurations. On average, the normalized standard deviation of the mean completion delays was only 15% (0.15 in Table 5.14) of a graph's period. Compare this to 37%, 99%, and 93% under SCHED\_OTHER, SCHED\_FIFO, and LITMUS<sup>RT</sup> without GPUSync, respectively. Other GPUSync configurations where  $\rho \in \{3, 6\}$  were competitive, but none performed as well as  $(3, 2, \text{FIFO})$ .

The above tables give us insight into worst-case and average-case behaviors of the tested system configurations. However, the distribution of observed completion delays is somewhat obscured. To gain deeper insights into these distributions, we plot the PDF of normalized completion delays in Figures 5.10 through 5.20. Each plot is derived from a histogram with a bucket width of 0.005. In each of these figures, the  $x$ -axis denotes a normalized completion delay. The  $y$ -axis denotes a probability density. To determine the probability that a normalized completion delay falls within the domain  $[a, b]$ , we sum the area under the curve between  $x = a$  and  $x = b$ ; the total area under each curve is 1.0. Generally, distributions with the greatest area near  $x = 1.0$  are best.

Our goal is to understand the *shape* of completion delay distributions, so each distribution is plotted on the same domain and range. We also *clip* the domain at  $x = 2.5$ , so the long tails of these distributions are not depicted. However, we have examined worst-case behaviors in the prior tables, so we do not revisit the topic here. We make several observations.

**Observation 48.** *The PDFs for GPUSync with  $\rho = 3$  show that normalized completion delays are most likely near 1.0.*

We see this in Figures 5.17 and 5.18 for GPUSync with  $\rho = 3$  for FIFO and PRIO engine locks, respectively. This result is not surprising, given prior Observation 45. However, in these PDFs we also see that they closely resemble the curve of a normal distribution, more so than any PDF of the other configurations. The PDFs for GPUSync with  $\rho = 6$  (Figures 5.19 and 5.20), and even the PDF for the SCHED\_OTHER configuration (Figure 5.10), have similar shapes, but they are not as strongly centered around 1.0. Also, they do not exhibit the same degree of symmetry.

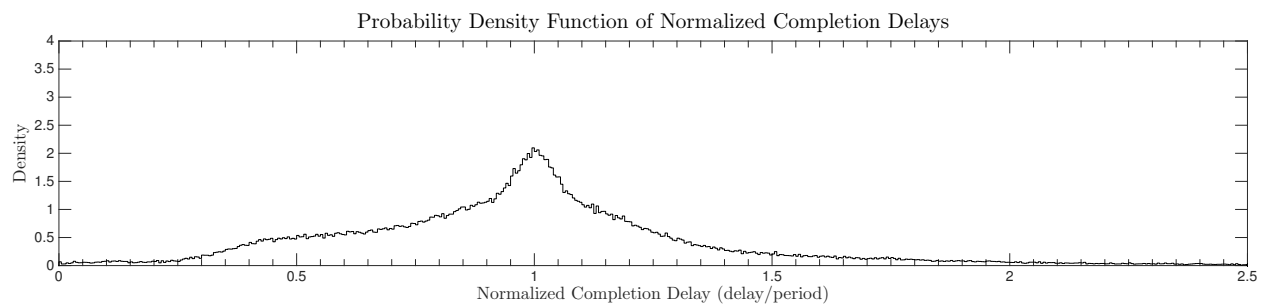


Figure 5.10: PDF of normalized completion delay data for SCHED\_OTHER.

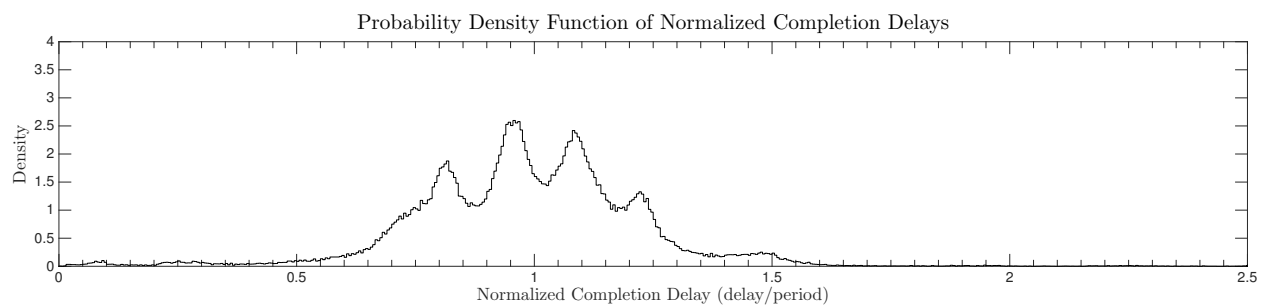


Figure 5.11: PDF of normalized completion delay data for SCHED\_FIFO.

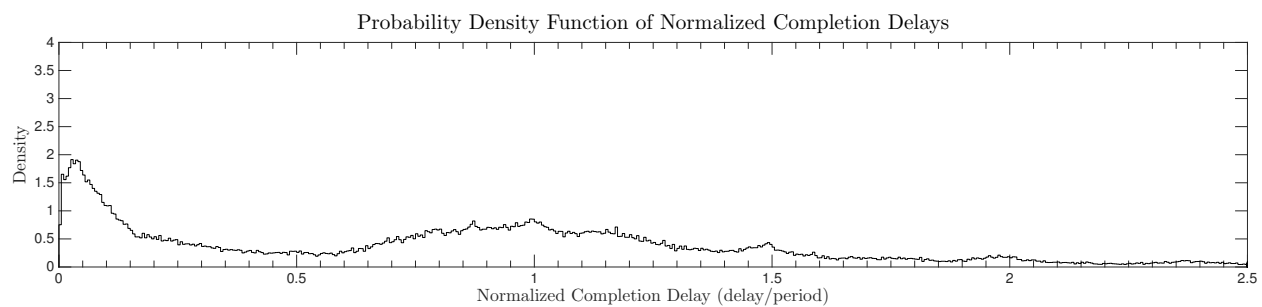


Figure 5.12: PDF of normalized completion delay data for LITMUS<sup>RT</sup> without GPUSync.

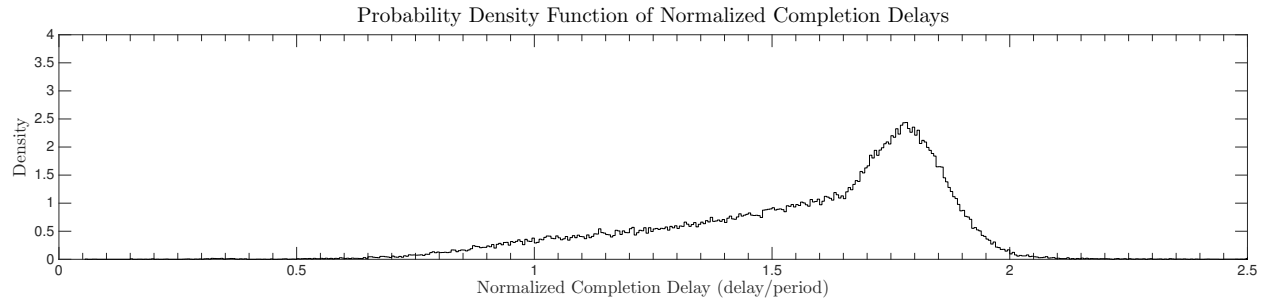


Figure 5.13: PDF of normalized completion delay data for GPUSync with (1,6,FIFO).

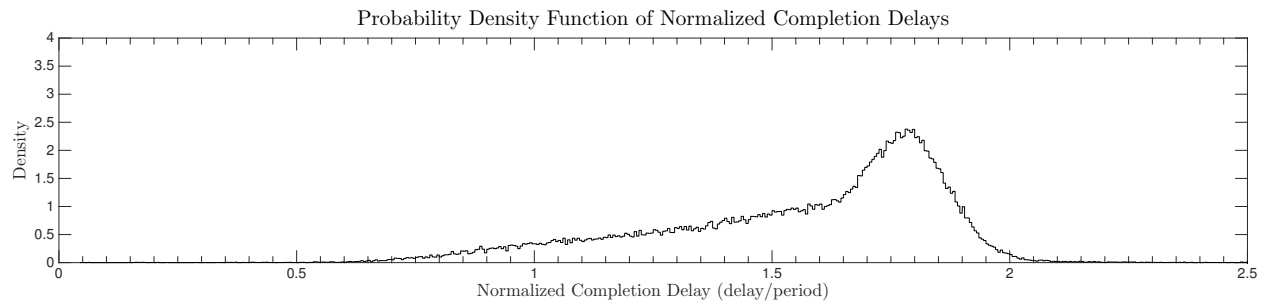


Figure 5.14: PDF of normalized completion delay data for GPUSync with (1,6,PRIO).

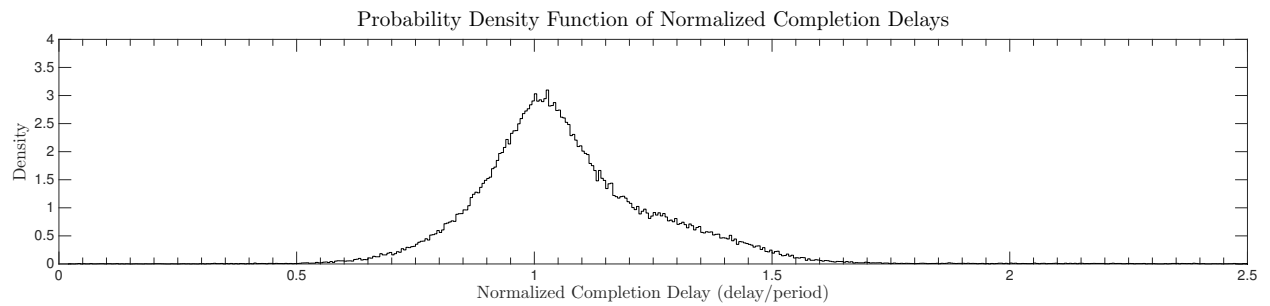


Figure 5.15: PDF of normalized completion delay data for GPUSync with (2,3,FIFO).

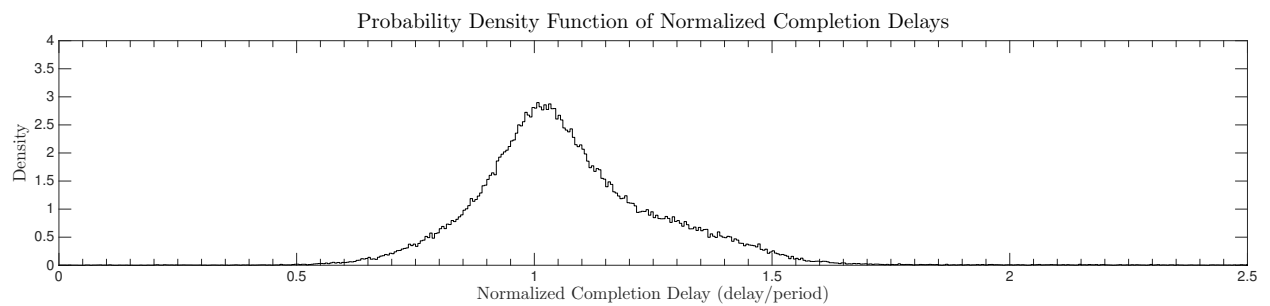


Figure 5.16: PDF of normalized completion delay data for GPUSync with (2,3,PRIO).

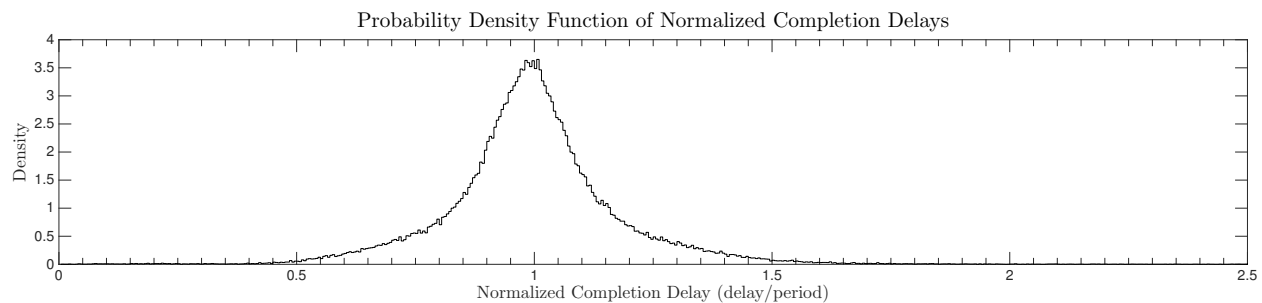


Figure 5.17: PDF of normalized completion delay data for GPUSync with (3, 2, FIFO).

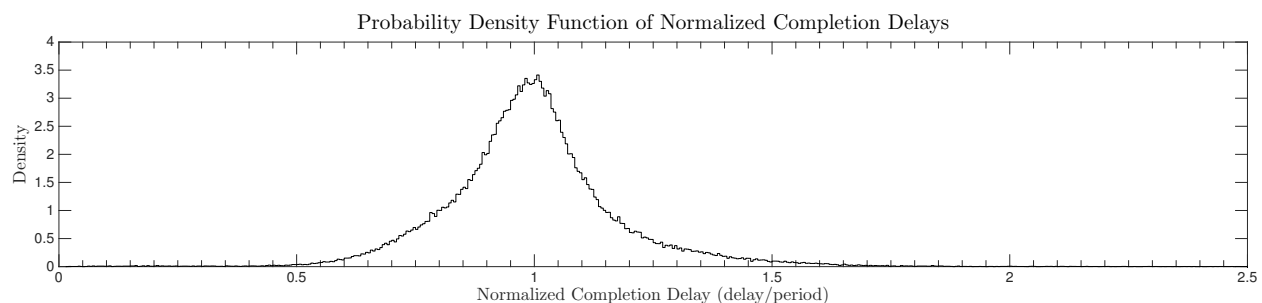


Figure 5.18: PDF of normalized completion delay data for GPUSync with (3, 2, PRIO).

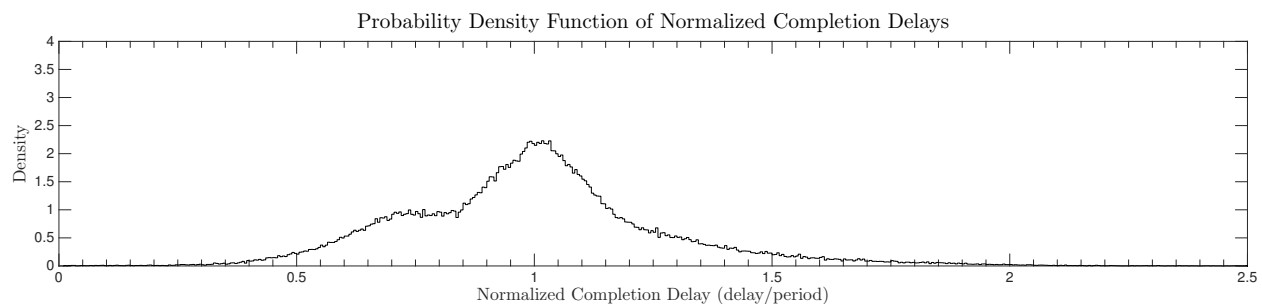


Figure 5.19: PDF of normalized completion delay data for GPUSync with (6, 1, FIFO).

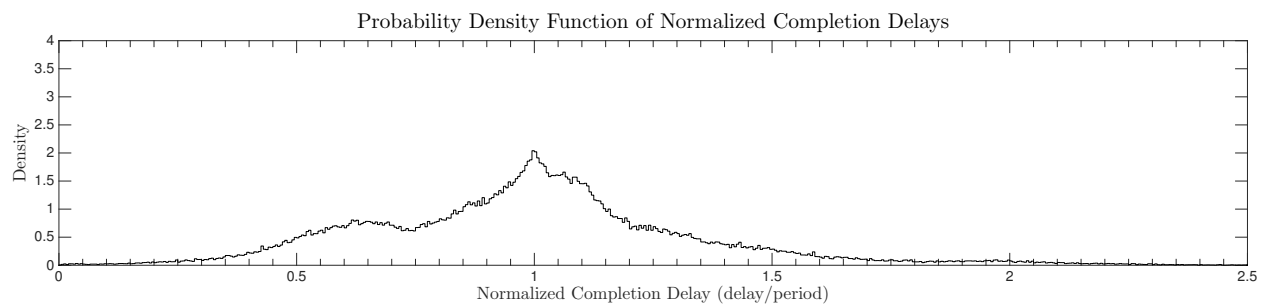


Figure 5.20: PDF of normalized completion delay data for GPUSync with (6, 1, PRIO).

Another characteristic of the PDFs for  $\rho = 3$  is that they are clearly unimodal. This is unlike the PDF in Figure 5.11 for the SCHED\_FIFO configuration, which has at least four distinct modes (indicated by the four peaks in the PDF), or the PDF in Figure 5.12 for LITMUS<sup>RT</sup> without GPUSync, which appears to be bimodal.

**Observation 49.** *The PDF for LITMUS<sup>RT</sup> without GPUSync suggests bursty completion behaviors.*

Figure 5.12 depicts the PDF for the LITMUS<sup>RT</sup> without GPUSync configuration. The PDF appears to have two modes. One mode is near  $x = 0.05$ ; the other is centered around 1.0. Not depicted in this figure is the long tail of the PDF.

When a job of a sink node of the video stabilization graph completes late, work can “back up” within the graph. Under the PGM early releasing policy of non-source nodes, a sink node with backed-up work may complete several jobs in quick succession as it catches up. In other words, one or more short completion delays may follow a very long completion delay. The first mode (the one near  $x = 0.05$ ) may indicate such a behavior. The corresponding long completion delays make up the long tail of the PDF, which we have clipped at  $x = 2.5$ .

Although the LITMUS<sup>RT</sup> without GPUSync configuration did not drop any frames, the playback of the stabilized video is far from smooth.

**Observation 50.** *The PDFs of the GPUSync configurations with  $\rho \in \{1, 2\}$  indicate unschedulability.*

Figures 5.13 through 5.16 depict the PDFs for GPUSync configurations where  $\rho \in \{1, 2\}$ . We see in these distributions that the majority of normalized completion delays are greater than 1.0. We may expect to see this characteristic in a PDF of an unschedulable configuration, as it indicates that completion delays are continually greater than task period (*i.e.*, greater than 1.0 after normalization), so deadlines are missed by continually greater margins. In this experiment, this ultimately results in dropped frames when the experiment terminates after 400 seconds. This is most clearly demonstrated in the PDFs for GPUSync configurations with  $\rho = 1$  in Figures 5.13 and 5.14—the bulk of normalized completion delays occur on the domain  $[1.5, 2]$ .

We also see this characteristic in Figures 5.15 and 5.16 for GPUSync configurations with  $\rho = 2$ , although it is less pronounced. In Figure 5.15, observe that the descent from the peak on the right of  $x = 1.0$  is more gradual than the ascent to the left of 1.0—more normalized completion delays are greater than 1.0. We see the same trend in Figure 5.16.

This concludes our examination observed completion delays.



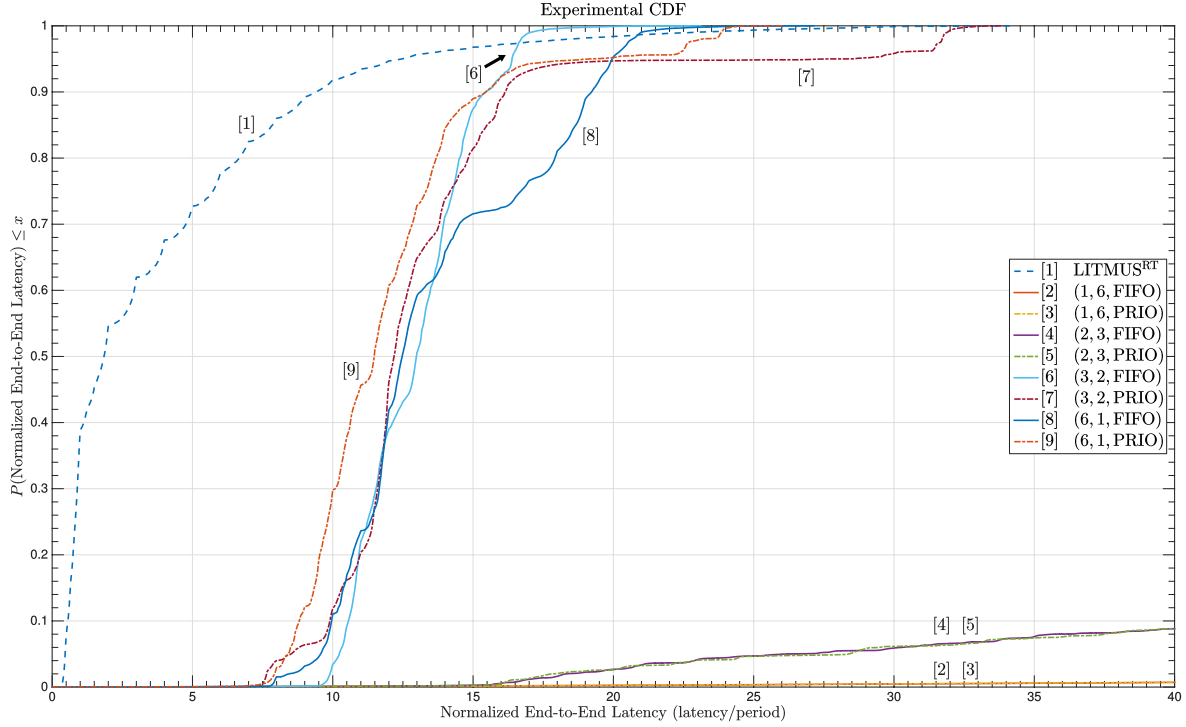


Figure 5.21: CDF of normalized observed end-to-end latency (domain clipped at  $x = 40$ ).

### 5.5.3.2 End-to-End Latency

We now examine the observed end-to-end response time latency of graphs under a subset of our various system configurations. Unlike in our study of completion delays, we used LITMUS<sup>RT</sup>'s low-level tracing capabilities to accurately record the release and completion time of source and sink node jobs, respectively. We compute the end-to-end latency of a single end-to-end execution of a graph by computing the difference between the release time of the source job and the completion time of the corresponding sink job. Due to our reliance on LITMUS<sup>RT</sup> for gathering these measurements, we limit our investigation here to GPUSync configurations and LITMUS<sup>RT</sup> without GPUSync. That is, we do not study end-to-end latencies under the SCHED\_OTHER or SCHED\_FIFO configurations.

We used the same experimental setup as before: The seventeen graphs described in Table 5.2 were executed for a duration of 400 seconds on six CPU cores and one GPU. The data we present in this section was gathered in a separate batch of experiments from the ones we performed to gather completion delay measurements, so there may be some discrepancy in worst-case measurements between the two data sets. However, as we shall see, we observe similar trends in both sets of measurements.

As with completion delays, end-to-end latencies are influenced by graph period. In order to study the end-to-end latencies of graphs with different periods, we *normalize* each measurement by dividing each measured end-to-end latency by the graph period. For every tested system configuration, we obtained a collection of normalized end-to-end latencies. Figure 5.21 plots the cumulative distribution function (CDF) for the observations of each tested configuration. These curves plot the likelihood that a given normalized end-to-end latency is less than a given value. For example, for the GPUSync configuration (3,2,PRIO) (curve 7), we see that approximately 95% of normalized end-to-end latency were less than 25. To make this observation, we find the  $y$ -value of curve 7 at  $x = 25$ .

In general, a curve that tends most towards the top-left corner of the figure is considered “best,” as this indicates that most end-to-end latencies are short. However, since we are most interested in *worst-case behavior* and *predictability*, we look for other characteristics in the curves. Good worst-case behavior is indicated by a curve with a short tail, *e.g.*, one where  $y = 1$  for a small  $x$ -value. Good predictability is indicated by a large increase in a curve over a short  $x$ -interval, as this means that there is little variance among many observations. Correspondingly, gradually increasing curves indicate a high degree of variance in the observations.

In order to study the characteristics of the more interesting curves, we have clipped the domain of Figure 5.21 to  $x = 40$ . This significantly truncates the curves for the GPUSync configurations where  $\rho \in \{1, 2\}$  (curves 2 through 5). However, we will study the end-to-end latencies of these configurations with a method better suited to studying curves with long tails, shortly. We make the following observations.

**Observation 51.** *GPUSync with (3,2,FIFO) exhibits the least variance in end-to-end latency.*

To make this observation, for each curve plotted in Figure 5.21, find the approximate normalized end-to-end latency (on the  $x$ -axis) where  $P(x)$  (on the  $y$ -axis) is first greater than zero. Let us denote this point with the variable  $a$ . Next, find the approximate normalized end-to-end latency where  $P(x)$  is nearly one. Let us denote this point with the variable  $b$ . Most, if not all, normalized end-to-end latencies for the curve lie within the domain of  $[a, b]$ , which is  $(b - a)$  units in length. For GPUSync with (3,2,FIFO) in curve 6, this domain is roughly  $[9, 20]$ , with a length of about 11 units. Compare this to LITMUS<sup>RT</sup> without GPUSync in curve 1, where the domain is  $[0.25, 30]$  (29.75 units long). GPUSync with (3,2,PRIO) (curve 7), the domain is about  $[7.5, 32.5]$  (25 units long). GPUSync with (6,1,FIFO) (curve 8), the domain is about  $[7.5, 22.5]$  (15 units long). GPUSync with (6,1,PRIO) (curve 9), the domain is about  $[7.5, 24.5]$  (17 units long). The domains for

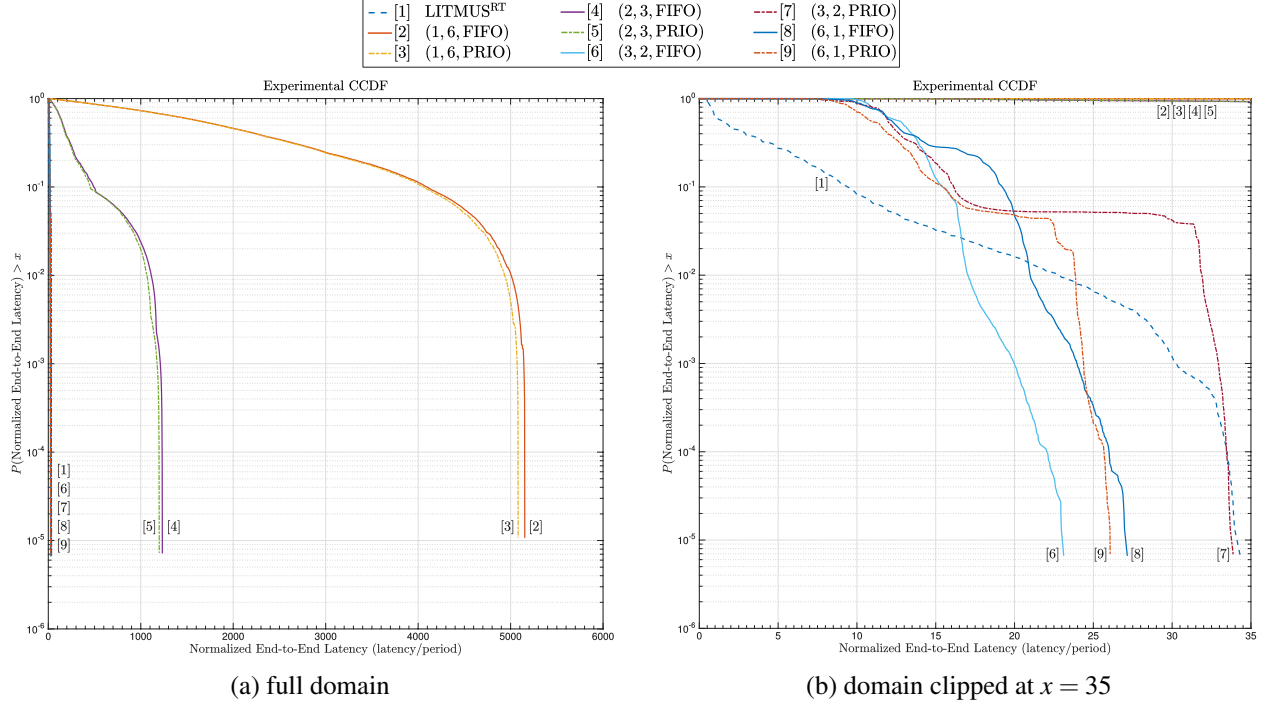


Figure 5.22: CCDF of normalized observed end-to-end latency (y-axis on log scale).

the remaining GPUSync configurations (curves 2 through 5) cannot be observed in Figure 5.21 due to the clipping the  $x$ -axis at  $x = 50$ .

**Observation 52.** *GPUSync configurations where  $\rho \in \{3, 6\}$  exhibit similar behavior for approximately 70% of normalized end-to-end latencies.*

In Figure 5.21, there is a clear clustering of curves 5 through 9 on the domain  $[7.5, 14]$ . For instance, for curves 5 through 8 normalized end-to-end latencies are below 13.5 about 35% of the time. It is not until  $y = 0.7$  (70%) that these curves really begin to differentiate. This is not to say that the curves for GPUSync configurations where  $\rho \in \{3, 6\}$  are indistinguishable before  $y = 0.7$ . For example, GPUSync configuration (6, 1, PRIO) initially shows the best end-to-end latency behavior, as its curve (curve 9) is above the other GPUSync configurations until about  $x = 15$ .

To study the worst-case end-to-end latency behavior of the system configurations in our experiment, we plot the *complementary* cumulative distribution function (CCDF) of our normalized observations of each tested configuration in Figure 5.22. In Figure 5.22(a), we plot our data on an  $x$ -axis long enough to capture all observations. In Figure 5.22(b), we plot the same data, but we clip the domain at  $x = 35$  in order to better depict the shape of the CCDFs for the better-performing configurations.

The CCDF is useful for determining the probability that a given observation is *greater* than a given  $x$  value. In other words, the CCDF is useful for studying worst-case behavior. This is especially true when the CCDF is plotted on a logarithmic  $y$ -axis, as we have done in Figure 5.22. For example, in Figure 5.22(b), find where curve 9 crosses the horizontal line for  $y = 10^{-1}$ . This occurs at about  $x = 15$ . This indicates that 10% of normalized end-to-end latencies for curve 9 are greater than 15. We may make similar types of observations along the horizontal lines for  $y = 10^{-2}$  (1%),  $y = 10^{-3}$  (0.1%),  $y = 10^{-4}$  (0.01%), and  $y = 10^{-5}$  (0.001%). We make the following observations.

**Observation 53.** *The normalized end-to-end latencies for GPUSync configurations are  $\rho \in \{1, 2\}$  are very poor.*

In Figure 5.22(a), curves 2 through 5 (those for GPUSync configurations where  $\rho \in \{1, 2\}$ ) extend beyond a normalized end-to-end latency of 1,000. This result should not be surprising, given the poor performance we observed for these configurations in our study of completion delays (Observations 42 and 50)—our video stabilization task set is clearly unschedulable under these configurations of GPUSync.

**Observation 54.** *GPUSync configurations with FIFO-ordered engine locks exhibit a smooth degradation in performance.*

In Figure 5.22(b), we observe that the curves for GPUSync with (3, 2, FIFO) (curve 6) and (6, 1, FIFO) (curve 8) descend gradually, in comparison to the priority-ordered engine lock configurations (curves 7 and 9). This indicates that performance degrades more smoothly under FIFO-ordered engine locks.

**Observation 55.** *GPUSync configurations with priority-ordered engine locks perform poorly for approximately 5% of the time, and they are prone to creating extreme outliers in end-to-end latencies.*

In Figure 5.22(b), observe the flat table-like trend for GPUSync configurations where priority-ordered engine locks are used (curves 7 and 9). These trends begin around  $x = 17$  at  $y = 5\%$ . The trend lasts until about  $x = 22$  and  $x = 31.5$  for GPUSync configurations with (6, 1, PRIO) (curve 9) and (3, 2, FIFO) (curve 7), respectively. After these points, the curves begin to drop precipitously.

These flat trends indicate a performance gap, where the majority of end-to-end graph invocations have good end-to-end latencies, while a few end-to-end graph invocations have very poor end-to-end latencies, relatively speaking. That is, these GPUSync configurations with priority-ordered locks are prone to extreme outlier behavior. Let us examine curve 7 for GPUSync with (3, 2, PRIO) more closely. Here, approximately

93% of end-to-end graph invocations have a normalized end-to-end latency *less* than 17. *Only* 3% of end-to-end graph invocations have a normalized end-to-end latency between 17 and 31.5—a very long interval of 14.5 units! The remaining 4% of end-to-end graph invocations have a normalized end-to-end latency greater than 31.5—this occurs over short interval from 31.5 to 34 (an interval of 2.5 units). We may make similar observations for GPUSync with (6, 1, PRIO) (curve 9), although outlier behavior is less extreme.

**Observation 56.** *GPUSync with (3, 2, FIFO) exhibits the best worst-case behavior.*

In Figure 5.22(b), find where each depicted curve intersects with the vertical line where normalized observed end-to-end latencies are 20 units ( $x = 20$ ). We find that only 0.09% of normalized observed end-to-end latencies for GPUSync with (3, 2, FIFO) (curve 6) are greater than 20. Contrast this to about 5% for the other GPUSync configurations where  $\rho \in \{3, 6\}$  (curves 7 through 9), or about 1.5% for LITMUS<sup>RT</sup> without GPUSync (curve 1). The differences are even more dramatic where normalized observed end-to-end latencies are 23. We find that only 0.002% of normalized observed end-to-end latencies for GPUSync with (3, 2, FIFO) (curve 6) are greater than 23. Contrast this to 0.25% for GPUSync with (6, 1, FIFO) (curve 8), 1% for LITMUS<sup>RT</sup> without GPUSync (curve 1), 2.5% for GPUSync with (6, 1, PRIO) (curve 9), and 5% for GPUSync with (3, 2, PRIO) (curve 7).

**Observation 57.** *GPUSync with (3, 2, FIFO) exhibits the best real-time behavior.*

The observation follows from Observations 51 and 56. The GPUSync configuration with (3, 2, FIFO) exhibits both the least variance in normalized end-to-end latencies, as well as the best worst-case behavior.

We make one last general observation before concluding our examination of the real-time runtime performance of our video stabilization task set.

**Observation 58.** *The behaviors we observed through the study of normalized end-to-end latencies correlate to those we made through the study of completion delays.*

In Section 5.5.3.1, we studied the performance of the video stabilization task set through completion delay metrics. As we discussed earlier, completion delay metrics are “fuzzy” in that we may observe a range of values for a task set that meets even hard real-time constraints (recall the discussion of Figure 5.9). Nevertheless, through completion delay metrics, we observed the following: **(i)** the video stabilization task set was clearly unschedulable under GPUSync configurations with  $\rho \in \{1, 2\}$ ; **(ii)** GPUSync configurations

with FIFO-ordered engine locks exhibited less variance in comparison to priority-ordered counterparts; (iii) GPUSync configurations with  $\rho = 3$  yielded better real-time behaviors than  $\rho = 6$ ; and (iv) the GPUSync configuration with (3,2,FIFO) performed best among all tested configurations. Each of these observations are supported by observations we make in this section based upon end-to-end latency data. This speaks towards the validity and usefulness of completion delay-based metrics.

## 5.6 Conclusion

Real-time applications with graph-based software architectures represent an important segment of computing that is not directly supported by the periodic or sporadic task models. As code complexity increases and heterogeneous computing platforms become more common and varied, we may expect to see such graph-based applications to become more common. This claim is supported by the recently ratified OpenVX standard and NVIDIA’s development of VisionWorks.

In this chapter, we extended GPUSync to support these graph-based real-time applications. We developed PGM<sup>RT</sup>, a middleware library, to track data dependencies among graph nodes and coordinate their execution. We also enhanced operating system support for graph-based real-time applications with modifications to LITMUS<sup>RT</sup> to dynamically adjust job release times and deadlines in accordance to real-time theory. Although PGM<sup>RT</sup> may be configured to support any POSIX-compliant platform, it may also be configured to tightly integrate with LITMUS<sup>RT</sup> to reduce system overheads and mitigate priority inversions.

We applied GPUSync and PGM<sup>RT</sup> to real-world software: an alpha version of NVIDIA’s VisionWorks. Through the use of PGM<sup>RT</sup>, we enhanced the back-end of VisionWorks to support multi-threaded pipelined execution (which is expressly *not* supported according to the OpenVX standard upon which VisionWorks is based). Due to the sheer amount of code and complexity, we could not feasibly modify VisionWorks to use GPUSync directly. Instead, we developed an interposition library to the CUDA runtime, libgpui. Libgpui enabled us to intercept all relevant CUDA-runtime calls made by VisionWorks and schedule them under GPUSync. The benefits offered by libgpui came at the expense of larger GPUSync token critical sections than strictly necessary. We then evaluated the real-time performance of our real-time version of VisionWorks under several Linux and LITMUS<sup>RT</sup>-based configurations, including eight unique configurations of GPUSync. Our results show that real-time GPU scheduling is necessary in order to achieve predictable real-time performance. We also identified which configurations of GPUSync gave the best real-time performance within the context

of our experiments. Importantly, we identified that there is a balance to be struck between GPU engine parallelism and the interference that *too much* parallelism may introduce.

## CHAPTER 6: CONCLUSION

At the beginning of this dissertation, we made the case that, relative to conventional CPUs, GPUs have the potential to increase a platform’s computational capacity by orders of magnitude, while remaining energy-efficient and affordable. These characteristics make GPUs an attractive computing platform for embedded systems with size, weight, and power constraints that still require a high-degree of computational capacity. These systems are perhaps best represented by today’s emerging advanced driver assistance systems and future autonomous vehicles. However, we also showed that GPUs cannot be used “as-is” if real-time predictability is required. This is due to a number of complicating factors raised by GPU management software and even the GPU hardware itself.

The main purpose of the research presented in this dissertation was to show that limitations in GPU technology *can* be overcome, and that this can be done without a ground-up reimplementing of GPU management software or crippling GPU hardware functionality. To demonstrate this, we designed and implemented GPUSync; extended overhead-aware evaluation methodologies to incorporate GPUSync’s mechanisms and GPU-related overheads; conducted a large-scale study to investigate the theoretical real-time performance of twelve different GPUSync configurations under a variety of workloads and overhead conditions; and performed two additional studies to evaluate observed runtime performance, the last of which applied GPUSync to a real-world computer vision software suite. In the following, we first summarize our results (Section 6.1), we then discuss avenues for future work (Section 6.2), and finally, conclude (Section 6.3).

### 6.1 Summary of Results

In Chapter 1, we presented the following thesis statement.

*The computational capacity of a real-time system can be greatly increased for data-parallel applications by the addition of GPUs as co-processors, integrated using real-time scheduling and synchronization techniques tailored to take advantage of specific GPU capabilities. Increases in*



*computational capacity outweigh costs, both analytical and actual, introduced by management overheads and limitations of GPU hardware and software.*

To support this thesis, we have developed the GPU scheduling framework GPUSync and we have investigated its performance in depth through theoretical and runtime performance experiments. We now describe the contributions made during the pursuit of this research.

**Survey of GPUs and similar co-processor accelerators for embedded applications.** In Chapter 2, we surveyed the current state of GPUs and similar co-processor accelerators that are targeted to embedded applications. The performance and cost characteristics of these embedded GPUs vary greatly. On one end of the spectrum, there are expensive high-performance military-grade GPUs used to perform radar, sonar, and computer vision processing for air, land, and sea autonomous vehicles. On the other end of the spectrum, there are cheap lower-performance GPUs that are integrated on-die with ARM processors, the majority of which are found in tablets and smartphones. Between these extremes, we find GPUs with mid-range performance capabilities that are targeted to automotive applications. Due to size, weight, and power constraints, these GPUs are also integrated on-die with ARM processors. Also within this space, we find specially designed co-processors aimed at providing a more efficient alternative to GPUs. These gains in efficiency are realized by sacrificing some the programmability and/or innate graphical capabilities of GPUs. By and large, these co-processors are designed to handle computer vision or video processing workloads.

Common to all of these GPUs and co-processors is a lack of proven real-time scheduling techniques that can integrate them with real-time CPU schedulers, while simultaneously addressing the unique constraints of the GPU or co-processor. This survey highlights the timeliness of this dissertation: embedded GPU and co-processor accelerators exist today, but we lack knowledge of how to best use them in a real-time system.

**Literature review of GPU scheduling techniques.** Also in Chapter 2, we surveyed GPU scheduling techniques developed by researchers and industry to modify a GPU’s default scheduling policies. We examined work from both real-time and non-real-time domains. Because this field is relatively new, our survey of techniques is nearly comprehensive, especially with respect to the real-time domain.

In the real-time domain, we identified two categories of scheduling techniques: persistent low-latency GPU kernels, and GPU resource scheduling. In the former, no scheduling takes place: a single persistent GPU kernel continuously polls for and performs work. This allows the GPU to be used in a single low-latency

application. In the latter category, competing real-time operations are scheduled to GPU processors to meet timing requirements (*i.e.*, deadlines). The work of this dissertation falls into this category.

In the non-real-time domain, we reviewed techniques for improving “fairness” of GPU resource allocation to competing tasks. Although similar to techniques for GPU resource scheduling in the real-time domain, fairness-oriented schedulers focus on achieving fairness on prolonged timescales. In contrast, a real-time scheduler may focus on more immediate timescales (*e.g.*, the next task deadline). Also in the non-real-time domain, we examined methods for consolidating work on a single GPU to maximize GPU utilization. This consolidation may take the form of GPU virtualization, which allows a GPU to be shared by competing virtual machines. Consolidation may also take the form of “GPGPU compute servers” that host GPGPU operations through remote-procedure-call-like methods. The clients of these servers may execute on the same machine as the server or remotely in a distributed system.

We examined the commonalities among the above techniques of both the real-time and non-real-time domains. In general, each technique, real-time or not, maybe classified as being either “API-driven” or “command-driven.” We considered the tradeoffs between these approaches in Chapter 2. We further studied API-driven GPU schedulers in Chapter 3, where we presented a taxonomy of eight general forms that an API-driven GPU scheduler may take. We discussed the tradeoffs among them.

**Design and implementation of GPUSync.** In Chapter 3, we presented the primary contribution of this dissertation: GPUSync. Unlike prior research, GPUSync takes a synchronization-based approach to GPU scheduling that simultaneously promotes affinity among computational tasks and GPUs, and fully exposes the parallelism offered by modern GPUs. While others have approached GPU management from a scheduling perspective, GPUSync’s synchronization-based techniques allow it to be “plugged in” to a variety of real-time schedulers (*e.g.*, partitioned, clustered, and global CPU schedulers). This synchronization-based approach allows GPUSync to be a “wrapper” around existing GPGPU runtimes and GPU device drivers. Thus, GPUSync may be used with closed-source GPU management software produced by GPU manufacturers. Such software usually offers better performance and a more robust set of features compared to reverse-engineered open-source alternatives. GPUSync also supports three budget enforcement policies to help prevent aberrant tasks from exceeding provisioned GPU execution times and to mitigate the negative effects on other tasks when they do.

In addition to addressing the core aspects of GPU scheduling, GPUSync also tackles issues that arise from working with real-world software. These issues *must* be resolved in order to avoid unbounded priority inversions due to GPU interrupt processing and GPGPU runtime callback threads—a fact often overlooked in prior research. To schedule interrupt processing, GPUSync intercepts GPU interrupts and redirects their processing to dedicated per-GPU threads. To schedule callback threads, GPUSync transparently determines which threads in a process are callback threads and automatically associates their scheduling priorities with those of the real-time tasks within the process. For both interrupt and callback threads, GPUSync uses priority inheritance mechanisms to ensure proper scheduling priorities to bound and minimize the duration of priority inversions.

Our implementation of GPUSync, which is over 20,000 lines of code, is freely available as patch to the LITMUS<sup>RT</sup> real-time operating system.<sup>1</sup>

**Quantification of GPU-related overheads.** In Chapter 4, we examined system overheads related to GPU processing. We categorized these overheads as either algorithmic or memory-related. Algorithmic overheads include those related to GPU interrupt processing. Through low-level instrumentation of our evaluation platform, we found that GPU interrupt processing times are usually relatively short (on the order of tens of microseconds), but we also found that, on the order of roughly one out of a million, interrupts may sometimes require over a millisecond to process. Although such occurrences are rare, the effects of interrupt processing must be considered in the design of a real-time system.

We also examined memory-related overheads, as they pertain to the effects of DMA operations between main system memory and GPU-local memory. We confirmed trends reported by others on DMA operation execution times when such operations are carried out on an idle system. We extended prior work and reported on DMA operation time for a system under worst-case bus contention scenarios. We also quantified increases in DMA operation execution time due to NUMA main memory, which has not been considered in prior work.

We also demonstrated that DMA operations may affect the execution time of any task executing on a CPU. The execution time cost of a main memory access by a CPU due to a cache-miss can increase significantly, since DMA operations may put load on the main memory bus, which is shared with system CPUs. We quantified these penalties, and we showed that DMA operations can increase cache-related overheads by as much as a factor of four on our evaluation platform. These cache overheads must be considered in

---

<sup>1</sup>GPUSync is available at [www.github.com/GElliott](http://www.github.com/GElliott).

overhead-aware schedulability analysis. Moreover, they can affect the choice a system designer may make when choosing between partitioned, clustered, or global CPU schedulers, as each scheduler class has a different sensitivity to cache-related overheads in terms of schedulability.

**Blocking bounds and overhead-aware schedulability analysis for GPUSync.** Also in Chapter 4, we extended suspension-oblivious preemption-centric schedulability analysis to include GPU-related overheads and to model GPUSync resource requests (*i.e.*, blocking analysis). To integrate GPU-related overheads into analysis, we considered worst-case scenarios, incorporating costs for GPU interrupt processing and related thread scheduling into our analytical model.

We presented several methods for bounding the blocking time of GPUSync resource requests. We began with “coarse-grain” analysis to demonstrate that blocking bounds grow asymptotically with respect to the number of system CPUs, GPUs, and configurable GPUSync parameters. The underlying locking protocols of GPUSync are *optimal* under suspension-oblivious analysis, so these asymptotic bounds do not depend upon the number of tasks in the task set under analysis. Specifically, for FIFO-ordered engine lock configurations of GPUSync that do *not* support direct DMA-based (“peer-to-peer”) migration between GPUs, we showed that the number of competing GPU engine requests that may block a given task’s own request has an asymptotic bound of  $\mathcal{O}(\rho)$ , where  $\rho$  denotes a configurable cap on the maximum number of tasks that may use a given GPU simultaneously. We also showed that this asymptotic bound increases to  $\mathcal{O}(\rho \cdot g)$  for copy engine requests in GPUSync configurations that do support peer-to-peer migration, where  $g$  denotes the number of GPUs in a task’s assigned GPU cluster. Following this coarse-grain analysis, we then presented “fine-grain” analysis whereby blocking bounds are tightened through the consideration of task set characteristics. For GPUSync configurations that support peer-to-peer GPU migration, we presented blocking analysis based upon integer linear programming. This approach avoids overly pessimistic assumptions that would be made under more conventional fine-grain analysis. To analyze GPUSync token requests, we presented the application of optimal bounds developed in prior work to GPUSync. We showed that the number of GPU token requests that may block a task’s own token request is bounded by  $\mathcal{O}(\frac{c}{\rho \cdot g})$ , where  $c$  denotes the number of CPUs in a task’s assigned CPU cluster.

**Large-scale schedulability experiments to determine best GPUSync configurations.** In Chapter 4, we also presented the results from a set of large-scale schedulability experiments that we performed to help us determine the best GPUSync configurations for soft real-time systems under a variety of overhead cost

models. These experiments took roughly 85,000 CPU hours to complete on a university compute cluster. To analyze the results of these experiments, we developed a ranking methodology to collapse many thousands of schedulability graphs into a form that is much more succinct and allows trends to be more easily identified. We summarize the four most salient results of these experiments:

1. Configurations where CPUs are clustered and GPUs are partitioned are preferred.
2. Configurations where *both* CPUs and GPUs were clustered outperformed, or were competitive with, configurations where CPUs and GPUs were fully partitioned. This is an important result, since partitioned CPU scheduling with partitioned GPUs represents the industry-standard approach.
3. For clustered GPU configurations, support for peer-to-peer migration is preferred.
4. Despite overheads and pessimistic analysis, we showed that in terms of real-time schedulability, GPUs can increase the computing capacity of a given platform. In the context of our experiments, we found that eight GPUs frequently resulted in the equivalent computational capacity of over 60 CPUs, assuming that an individual GPU offered a conservative 32x speed-up with respect to a single CPU.

**Runtime experiments to validate performance benefits of GPUSync.** We also evaluated the runtime performance of GPUSync in Chapter 4. This runtime evaluation was in two parts: **(i)** through focused experiments, we quantified the performance of individual GPUSync aspects pertaining to the efficacy of budget enforcement mechanisms, the accuracy of migration cost prediction, and benefits of migration affinity heuristics; and **(ii)** we examined the high-level real-time properties of several GPUSync configurations by observing job response times of tasks that performed computer vision “feature tracking” calculations on GPUs.

In **(i)**, we demonstrated that GPUSync’s budget enforcement policies are effective at limiting long-term GPU resource utilization to provisioned constraints, even when tasks sporadically exceeded their GPU execution time budgets. Also in **(i)**, we demonstrated in the context of our experiments that GPUSync’s migration cost prediction and affinity heuristics can greatly reduce the frequency of GPU task migration; in most cases, migrations were avoided and the most costly “far” GPU migrations were practically eliminated.

In **(ii)**, our experiments demonstrated that clustered GPU configurations (with peer-to-peer migration) can outperform partitioned GPU configurations. In light of the results of our schedulability experiments, where we showed that partitioned GPUs are to be preferred, our result here is particularly informative: provided that

a system designer is willing to sacrifice some degree of schedulability, clustered GPU configurations may be the preferred platform configuration.

**Design and implementation of PGM<sup>RT</sup>.** In Chapter 5, we described PGM<sup>RT</sup>, a portable real-time dataflow scheduling middleware we designed and implemented to support graph-based real-time task models.<sup>2</sup> This middleware uses low-overhead mechanisms to coordinate the transmission of data among cooperating real-time tasks. Through integration with the underlying real-time operating system, and the careful ordering of the sequence of operations needed to perform data transmission, PGM<sup>RT</sup> minimizes or entirely avoids priority inversions that would arise in more naïve approaches. Although PGM<sup>RT</sup> offers the best real-time performance under LITMUS<sup>RT</sup>, it remains portable to POSIX-compliant operating systems.

**Design and implementation of a new pipelined, multi-threaded, execution model for OpenVX.** Also in Chapter 5, we examined a recently ratified open standard for computer vision workloads, OpenVX. OpenVX uses a graph-based software architecture, where computational primitive operations, represented by nodes, are linked together in a graph described by the programmer. We examined this standard, identified limitations in its prescribed runtime execution model and lack of real-time support, and proposed a new real-time-friendly pipelined execution model that maximizes the parallel execution of a graph’s nodes.

Leveraging GPUSync and PGM<sup>RT</sup>, we implemented our new execution model for OpenVX in Vision-Works, an OpenVX implementation by NVIDIA designed to use GPUs for most computations. Through a case study evaluation involving a complex computer vision workload, we demonstrated that our approach increases real-time predictability. Moreover, we showed that GPUSync offers significantly better performance over approaches that lack explicit real-time GPU management. The results of this study also proved that GPUSync is capable of supporting real-world real-time applications.

## 6.2 Future Work

We now discuss future work and new directions of research that could improve the results presented in this dissertation.

**Application of GPUSync design principles to non-GPU devices.** In this dissertation, we focused our attention on integrating GPUs into real-time systems. However, many of the techniques we developed herein

---

<sup>2</sup>PGM<sup>RT</sup> is available at [www.github.com/GElliott](http://www.github.com/GElliott).

may be applicable to non-GPU devices. Non-preemptive execution on devices such as FPGAs and DSPs are prime candidates. However, we can also apply these techniques to preemptive accelerators such as Intel's Xeon Phi. The techniques developed for GPUSync may also be applied to other I/O devices. For example, it could be used to manage network interconnects, such as InfiniBand network cards, that use DMA to shuttle data between applications and device memory.

**GPUSync and fixed-priority scheduling.** In Chapter 4, we examined the runtime performance of GPUSync under fixed-priority scheduling, specifically rate monotonic scheduling. However, we limited our investigation of *schedulability* to deadline-based scheduling (specifically, fair-lateness scheduling) because of this class of scheduler's better soft real-time capabilities (*e.g.*, bounded deadline tardiness) as support for soft real-time systems was a primary goal of this dissertation. Although GPUSync supports fixed-priority scheduling as it is designed today, it remains an open question as to which configurations offer the best schedulability under fixed-priority scheduling.

Of all possible fixed-priority GPUSync configurations, one configuration deserves the closest and most immediate examination: a platform with partitioned CPUs with partitioned GPUs shared among tasks assigned to different CPU partitions. The importance of this configuration is due to industry's current real-time system design practices. Commercial real-time operating systems usually only support fixed-priority scheduling. Moreover, partitioned CPU scheduling is common. In such systems where CPUs outnumber GPUs, developers will be tempted to share GPUs among CPU partitions. However, in Chapter 4, we saw that under fair-lateness scheduling, configurations where GPUs are shared among tasks assigned to different CPU partitions (or clusters) were among the *worst* evaluated configurations. Does this hold true under fixed-priority scheduling? The answer to this question would have a direct impact on fixed-priority systems being designed today.

**Platforms with heterogeneous GPUs.** In Chapter 3, we assumed that all system GPUs were homogeneous. This assumption simplified both the design and implementation of GPUSync, as well as the schedulability analysis we presented in Chapter 4. One easy way to extend GPUSync to these platforms today is to create smaller clusters of homogeneous GPUs from the set of heterogeneous GPUs. To support clusters of heterogeneous GPUs, migration cost heuristics must be adapted to incorporate GPUs with different performance characteristics. If peer-to-peer migration is to be supported among heterogeneous GPUs, then

schedulability analysis must be extended to consider task migration between GPUs with differing number of copy engines (*i.e.*, between a GPU with two copy engines and another with only one copy engine).

**Real-time scheduling of integrated GPUs.** When the work of this dissertation began, integrated GPUs capable of supporting GPGPU applications did not exist. Today, they are becoming increasingly common. GPUSync can support integrated GPUs. However, the framework may be unnecessarily complex for platforms with a single integrated GPU. As we discussed in Chapter 2, integrated GPUs lack copy engines, since they lack GPU-local memory. Instead, these GPUs use a portion of main system memory. As a result, a real-time GPU scheduling framework for integrated GPUs need not support DMA scheduling. GPUSync’s two-phase GPU resource allocation scheme (*i.e.*, the token allocator and engine locks) is harder to justify for these platforms. Instead, a simpler approach may be to do away with the token allocator and shift the side-effects of the token allocator (*i.e.*, its integration with GPUSync’s interrupt processing) to a singular execution engine lock. This configuration can be approximated by GPUSync today by setting the number of GPU tokens to be equal the number of GPU-using tasks. GPU interrupts would remain properly prioritized, since the thread responsible for processing GPU interrupts can only inherit the priority of a task blocked waiting for its own GPU operation to complete. However, this approach incurs needless overheads.

**Fine-grain scheduling of GPUs for graph-based task models.** In the blocking analysis of GPUSync presented in Chapter 4, we always assume that requests are blocked under worst-case scenarios, even if such occurrences are rare in practice. For example, in our runtime evaluation of clustered GPU configurations in Chapter 4, we observed that GPU migrations are rare. However, in analysis, we assume that *every* job migrates to a GPU. Moreover, since we use suspension-oblivious analysis, the time a job executes on a GPU engine is masked by fictitious CPU demand. These limitations result in a significant loss of actual CPU utilization.

In Chapter 5, we extended GPUSync support to a graph-based task model. However, GPU operations remained embedded within GPU-using tasks that executed on CPUs. It may be better to further break graph-based applications down into a finer degree of granularity, where each node executes entirely upon a single type of processor or engine. GPU engines would be managed by schedulers, instead of locking protocols, since locking protocols for GPU resource allocation are no longer necessary when a job executes entirely upon one type of processor. It then may be possible to apply the schedulability analysis developed in Elliott *et al.* (2014) to separately analyze the schedulability of each cluster. In theory, this approach allows full



resource utilization, in terms of both CPUs and GPU engines, while maintaining bounded deadline tardiness. However, it also requires near compiler-level insight into the execution patterns of application code in order to decompose a GPU-using task into a sequence of fine-grain CPU code segments and GPU operations.

### **6.3 Closing Remarks**

In this dissertation, we have shown how we may integrate GPUs with CPUs in a real-time system. This specific topic is only a part of a broader field of research on real-time scheduling of heterogeneous processors. As it becomes more difficult to increase the number of transistors in a processor, we can expect to see the continued specialization of computing elements to maximize the utility of each transistor. We speculate that the integration of GPUs onto CPU processor dies is only the first sign of a more significant trend towards further integration and specialization. These trends speak to the importance of further development of our understanding of real-time scheduling on heterogeneous platforms. We hope that the work presented in this dissertation will contribute to the foundation of future research into real-time GPU scheduling, and scheduling on heterogeneous processors in general.

## BIBLIOGRAPHY

- Adams, A., Jacobs, D. E., Dolson, J., Tico, M., Pulli, K., Talvala, E., Ajdin, B., Vaquero, D., Lensch, H., Horowitz, M., *et al.* (2010). The Frankencamera: an experimental platform for computational photography. *ACM Transactions on Graphics*, 29(4):29:1–29:12.
- Aila, T. and Laine, S. (2009). Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics*, pages 145–149.
- Amazon (2014). *Amazon Elastic Compute Cloud: User Guide for Linux*.
- AMD (2013). *OpenCL Programming Guide*. Revision 2.7.
- Anderson, J., Bud, V., and Devi, U. (2005). An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 199–208.
- Athow, D. (2013). ARM releases new GPU powerful enough to take on gaming consoles. <http://www.techradar.com/us/news/computing-components/processors/arm-releases-mali-t760-16-core-and-t720-8-core-gpu-1194951>.
- Aumiller, J., Brandt, S., Kato, S., and Rath, N. (2012). Supporting low-latency CPS using GPUs and direct I/O schemes. In *Proceedings of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 437–442.
- Baker, T. (1991). Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99.
- Baruah, S. (2004). Feasibility analysis of preemptive real-time systems upon heterogeneous multiprocessor platforms. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 37–46.
- Basaran, C. and Kang, K.-D. (2012). Supporting preemptive task executions and memory copies in GPGPUs. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 287–296.
- Bauer, S., Kohler, S., Doll, K., and Brunsmann, U. (2010). FPGA-GPU architecture for kernel SVM pedestrian detection. In *Proceedings of the 2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 61–68.
- Benenson, R., Mathias, M., Timofte, R., and Gool, L. V. (2012). Pedestrian detection at 100 frames per second. In *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2903–2910.
- Berezovskyi, K., Bletsas, K., and Andersson, B. (2012). Makespan computation for GPU threads running on a single streaming multiprocessor. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 277–286.
- Berezovskyi, K., Bletsas, K., and Petters, S. (2013). Faster makespan estimation for GPU threads on a single streaming multiprocessor. In *Proceedings of the 18th IEEE Conference on Emerging Technologies and Factory Automation*, pages 1–8.
- Berezovskyi, K., Santinelli, L., Bletsas, K., and Tovar, E. (2014). WCET measurement-based and extreme value theory characterisation of CUDA kernels. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, pages 279–288.

- Betts, A. and Donaldson, A. (2013). Estimating the WCET of GPU-accelerated applications using hybrid analysis. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 193–202.
- Block, A., Leontyev, H., Brandenburg, B., and Anderson, J. (2007). A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–56.
- Bourdev, L. and Brandt, J. (2005). Robust object detection via soft cascade. In *Proceedings of the 2005 IEEE Conference on Computer Vision and Pattern Recognition*, volume 2, pages 236–243.
- Bradley, T. (2012). *Hyper-Q Example*.
- Brandenburg, B. (2011a). SchedCAT. <http://mpi-sws.org/~bbb/projects/schedcat>.
- Brandenburg, B. (2011b). *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, University of North Carolina at Chapel Hill.
- Brandenburg, B. (2012). Virtually exclusive resources. Technical Report MPI-SWS-2012-005, Max Planck Institute for Software Systems.
- Brandenburg, B. (2014a). A synchronous IPC protocol for predictable access to shared resources in mixed-criticality systems. In *Proceedings of the 35th IEEE International Real-Time Systems Symposium*, pages 196–206.
- Brandenburg, B. (2014b). LITMUS<sup>RT</sup>: Linux testbed for multiprocessor scheduling in real-time systems. <http://www.litmus-rt.org>.
- Brandenburg, B. and Anderson, J. (2013). The OMLP family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Systems*, 17(2):277–342.
- Brandenburg, B., Leontyev, H., and Anderson, J. (2010). An overview of interrupt accounting techniques for multiprocessor real-time systems. *Journal of Systems Architecture*, 57(6):638–654.
- Brill, F. and Albuz, E. (2014). NVIDIA VisionWorks toolkit. Presented at the 2014 GPU Technology Conference.
- Buck, I. (2010). The evolution of GPUs for general purpose computing. Presented at the 2010 GPU Technology Conference.
- Calandrino, J., Leontyev, H., Block, A., Devi, U., and Anderson, J. (2006). LITMUS<sup>RT</sup>: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 111–126.
- Chen, C. and Tripathi, S. (1994). Multiprocessor priority ceiling based protocols. Technical Report CS-TR-3252, University of Maryland.
- Chen, M. (1992). *Schedulability analysis of resource access control protocols in real-time systems*. PhD thesis, University of Illinois at Urbana-Champaign.
- CogniVue (2014). CV220x image cognition processors. <http://www.cognivue.com/G2APEXIP.php>.
- Devi, U. and Anderson, J. (2006). Tardiness bounds under global EDF scheduling on a multiprocessor. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 330–341.

- Duato, J., Pena, A., Silla, F., Mayo, R., and Quintana-Ortí, E. (2010). rCUDA: Reducing the number of gpu-based accelerators in high performance clusters. In *Proceedings of the 2010 International Conference on High Performance Computing and Simulation*, pages 224–231.
- Easwaran, A. and Andersson, B. (2009). Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *Proceedings of the 30th IEEE International Real-Time Systems Symposium*, pages 377–386.
- Elliott, G. and Anderson, J. (2012a). The limitations of fixed-priority interrupt handling in PREEMPT\_RT and alternative approaches. In *Proceedings of the 14th OSADL Real-Time Linux Workshop*, pages 149–155.
- Elliott, G. and Anderson, J. (2012b). Robust real-time multiprocessor interrupt handling motivated by GPUs. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 267–276.
- Elliott, G. and Anderson, J. (2013). An optimal  $k$ -exclusion real-time locking protocol motivated by multi-GPU systems. *Real-Time Systems*, 49(2):140–170.
- Elliott, G. and Anderson, J. (2014). Exploring the multitude of real-time multi-GPU configurations. In *Proceedings of the 35th IEEE International Real-Time Systems Symposium*, pages 260–271.
- Elliott, G., Kim, N., Liu, C., and Anderson, J. (2014). Minimizing response times of automotive dataflows on multicore. In *Proceedings of the 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10.
- Elliott, G., Ward, B., and Anderson, J. (2013). GPUSync: A framework for real-time GPU management. In *Proceedings of the 34th IEEE International Real-Time Systems Symposium*, pages 33–44.
- Erickson, J. (2014). *Managing Tardiness Bounds and Overload in Soft Real-Time Systems*. PhD thesis, University of North Carolina at Chapel Hill.
- Free Software Foundation, Inc. (1991). Gnu general public license, version 2. <http://www.gnu.org/licenses/old-licenses/gpl-2.0.html>.
- Freescale (2014). *SCP220x ICP Family Data Sheet*.
- Fujii, Y., Azumi, T., Nishio, N., Kato, S., and Edahiro, M. (2013). Data transfer matters for GPU computing. In *Proceedings of the 19th International Conference on Parallel and Distributed Systems*, pages 275–282.
- Gai, P., Abeni, L., and Buttazzo, G. (2002). Multiprocessor DSP scheduling in system-on-a-chip architectures. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, pages 231–238.
- Gai, P., di Natale, M., Lipari, G., Ferrari, A., C.Gabellini, and Marceca, P. (2003). A comparison of MPCP and MSRP when sharing resources in the Janus multiple processor on a chip platform. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 189–198.
- General Electric (2011). *GPGPU COTS Platforms: High-Performance Computing Solutions*.
- Giunta, G., Montella, R., Agrillo, G., and Coviello, G. (2010). A GPGPU transparent virtualization component for high performance computing clouds. In *Euro-Par 2010: Parallel Processing*, volume 6271, pages 379–391. Springer Berlin Heidelberg.
- Goddard, S. (1998). *On the Management of Latency in the synthesis of real-time signal processing systems from processing graphs*. PhD thesis, University of North Carolina at Chapel Hill.

- Green, M. (2000). “How long does it take to stop?” Methodological analysis of driver perception-brake times. *Transportation Human Factors*, 2(3):195–216.
- Gupta, V., Gavrilovska, A., Schwan, K., Kharche, H., Tolia, N., Talwar, V., and Ranganathan, P. (2009). GVIM: GPU-accelerated virtual machines. In *Proceedings of the 3rd Workshop on System-level Virtualization for High Performance Computing*, pages 17–24.
- Gurobi Optimization, Inc. (2015). Gurobi optimizer reference manual. <http://www.gurobi.com>.
- Harris, M. (2009). Gpgpu.org. <http://gpgpu.org/about>.
- Harris, M. (2015). GPU pro tip: CUDA 7 streams simplify concurrency. <http://devblogs.nvidia.com/parallelforall/gpu-pro-tip-cuda-7-streams-simplify-concurrency>.
- Harrison, O. and Waldron, J. (2008). Practical symmetric key cryptography on modern graphics hardware. In *Proceedings of the 17th USENIX Security Symposium*, pages 195–209.
- Ho, J. and Smith, R. (2015). NVIDIA Tegra X1 preview and architecture analysis. *AnandTech*.
- Homm, F., Kaempchen, N., Ota, J., and Burschka, D. (2010). Efficient occupancy grid computation on the GPU with lidar and radar for road boundary detection. In *Proceedings of the 2010 IEEE Intelligent Vehicles Symposium*, pages 1006–1013.
- HSA Foundation (2014). *HSA Platform System Architecture*. Provisional 1.0.
- Intel (2014). Intel microprocessor export compliance metrics. <http://www.intel.com/support/processors/sb/cs-017346.htm>.
- Jeffay, K. and Goddard, S. (1999). A theory of rate-based execution. In *Proceedings of the 20th IEEE International Real-Time Systems Symposium*, pages 304–314.
- Jeffay, K. and Stone, D. (1993). Accounting for interrupt handling costs in dynamic priority task systems. In *Proceedings of the 14th IEEE International Real-Time Systems Symposium*, pages 212–221.
- Kato, S., Lakshmanan, K., Kumar, A., Kelkar, M., Ishikawa, Y., and Rajkumar, R. (2011a). RGEM: A responsive GPGPU execution model for runtime engines. In *Proceedings of the 32nd IEEE International Real-Time Systems Symposium*, pages 57–66.
- Kato, S., Lakshmanan, K., Rajkumar, R., and Ishikawa, Y. (2011b). TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the 2011 USENIX Annual Technical Conference*, pages 17–30.
- Kato, S., McThrow, M., Maltzahn, C., and Brandt, S. (2012). Gdev: First-class GPU resource management in the operating system. In *Proceedings of the 2012 USENIX Annual Technical Conference*, pages 401–412.
- Keller, J. (2013). NATO minehunting UUV relies on GPU-based embedded processor from GE for imaging sonar. *Military and Aerospace Electronics*.
- Khronos Group (2005). *OpenMAX IL API Specification*.
- Khronos Group (2014a). *The OpenCL Specification*. Version 2.0.
- Khronos Group (2014b). *The OpenGL Shading Language*. Version 4.40.

- Khronos Group (2014c). *The OpenVX™ Specification*. Version 1.0, Revision r28647.
- Kim, J., Andersson, B., Niz, D., and Rajkumar, R. (2013). Segment-fixed priority scheduling for self-suspending real-time tasks. In *Proceedings of the 34th IEEE International Real-Time Systems Symposium*, pages 246–257.
- Kim, J. S., Hwangbo, M., and Kanade, T. (2009). Realtime affine-photometric KLT feature tracker on GPU in CUDA framework. In *Proceedings of the 12th IEEE International Conference on Computer Vision Workshops*, pages 886–893.
- Kim, M. and Noble, B. (2001). Mobile network estimation. In *Proceedings of the 7th International Conference on Mobile Computing and Networking*, pages 298–309.
- King, G.-H., Cai, Z.-Y., Lu, Y.-Y., Wu, J.-J., Shih, H.-P., and Chang, C.-R. (2010). A high-performance multi-user service system for financial analytics based on web service and GPU computation. In *Proceedings of the 2010 International Symposium on Parallel and Distributed Processing with Applications*, pages 327–333.
- Klug, B. (2011). TI announces OMAP4470 and specs. <http://www.anandtech.com/show/4413/ti-announces-omap-4470-and-specs-powervr-sgx544-18-ghz-dual-core-cortexa9>.
- Kuhn, T., Kummert, F., and Fritsch, J. (2012). Spatial ray features for real-time ego-lane extraction. In *Proceedings of the 15th IEEE International Conference on Intelligent Transportation Systems*, pages 288–293.
- Lakshmanan, K., Niz, D., and Rajkumar, R. (2009). Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Proceedings of the 30th IEEE International Real-Time Systems Symposium*, pages 469–478.
- Lalonde, M., Byrns, D., Gagnon, L., Teasdale, N., and Laurendeau, D. (2007). Real-time eye blink detection with GPU-based SIFT tracking. In *Proceedings of the 4th Canadian Conference on Computer and Robot Vision*, pages 481–487.
- Lamastra, G., Lipari, G., and Abeni, L. (2001). A bandwidth inheritance algorithm for real-time task synchronization in open systems. In *Proceedings of the 22nd IEEE International Real-Time Systems Symposium*, pages 151–160.
- Lewandowski, M., Stanovich, M. J., Baker, T. P., Gopalan, K., and Wang, A. (2007). Modeling device driver effects in real-time schedulability analysis: Study of a network driver. In *Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Systems*, pages 57–68.
- Liu, C. (2013). *Efficient Design, Analysis, and Implementation of Complex Multiprocessor Real-Time Systems*. PhD thesis, University of North Carolina at Chapel Hill.
- Liu, C. and Anderson, J. (2010). Supporting soft real-time DAG-based systems on multiprocessors with no utilization loss. In *Proceedings of the 31st IEEE International Real-Time Systems Symposium*, pages 3–13.
- Liu, C. and Layland, J. (1973). Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 30:46–61.
- Liu, J. (2000). *Real-Time Systems*. Prentice Hall.

- Lopez, J., Diaz, J., and Garcia, D. (2004). Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28(1):39–68.
- Luebke, D., Harris, M., Krüger, J., Purcell, T., Govindaraju, N., Buck, I., Woolley, C., and Lefohn, A. (2004). GPGPU: General purpose computation on graphics hardware. In *ACM SIGGRAPH 2004 Course Notes*.
- Mangharam, R. and Saba, A. (2011). Anytime algorithms for GPU architectures. In *Proceedings of the 32nd IEEE International Real-Time Systems Symposium*, pages 47–56.
- Manica, N., Abeni, L., Palopoli, L., Faggioli, D., and Scordino, C. (2010). Schedulable device drivers: Implementation and experimental results. In *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 53–61.
- McKenney, P. E. (2009). “Real Time” vs. “Real Fast”: How to choose? In *Proceedings of the 11th OSADL Real-Time Linux Workshop*, pages 1–12.
- McMurray, I. (2011). GE announces high performance 360° local situational awareness visualization system. <http://www.ge-ip.com/news/ge-announces-high-performance-360-local-situational-awareness-visualization-system/n2917>.
- Menychtas, K., Shen, K., and Scott, M. L. (2014). Disengaged scheduling for fair, protected access to fast computational accelerators. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 301–316.
- Meuer, H., Strohmaier, E., Dongarra, J., and Simon, H. (2014). TOP500 supercomputing sites. <http://www.top500.org>.
- Microsoft (2014). Programming guide for HLSL. [http://msdn.microsoft.com/en-us/library/windows/desktop/bb509635\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/bb509635(v=vs.85).aspx).
- Miller, I., Campbell, M., Huttenlocher, D., Kline, F., Nathan, A., Lupashin, S., Catlin, J., Schimpf, B., Moran, P., Zych, N., *et al.* (2009). Team Cornell’s skynet: Robust perception and planning in an urban environment. *Journal of Field Robotics*, 56:257–304.
- Mok, A. (1983). *Fundamental design problems of distributed systems for the hard-real-time environment*. PhD thesis, Massachusetts Institute of Technology.
- Mussi, L., Cagnoni, S., Cardarelli, E., Daolio, F., Medici, P., and Porta, P. P. (2010). GPU implementation of a road sign detector based on particle swarm optimization. *Evolutionary Intelligence*, 3(3-4):155–169.
- Muyan-Ozcelik, P., Glavtchev, V., Ota, J. M., and Owens, J. D. (2011). Real-time speed-limit-sign recognition an embedded system using a GPU. *GPU Computing Gems*, pages 473–496.
- Nogueira, L. and Pinho, L. (2008). Shared resources and precedence constraints with capacity sharing and stealing. In *Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8.
- NVIDIA (2012). *Cg 3.1 Reference Manual*. Version 3.1.
- NVIDIA (2013). Chimera: The NVIDIA computational photography architecture.
- NVIDIA (2014a). Change log of the NVIDIA Linux x86\_64 346.22 device driver. [ftp://download.nvidia.com/XFree86/Linux-x86\\_64/346.22/NVIDIA-Linux-x86\\_64-346.22.run](ftp://download.nvidia.com/XFree86/Linux-x86_64/346.22/NVIDIA-Linux-x86_64-346.22.run).

- NVIDIA (2014b). *Multi-Process Service*.
- NVIDIA (2014c). *NVIDIA CUDA C Programming Guide*. Version 6.5.
- NVIDIA (2014d). *NVIDIA GRID Virtualization Solutions Guide*.
- NVIDIA (2014e). NVIDIA Quadro digital video pipeline. [http://www.nvidia.com/object/quadro\\_dvp.html](http://www.nvidia.com/object/quadro_dvp.html).
- NVIDIA (2014f). *Technical Reference Manual: NVIDIA Tegra K1 Mobile Processor*.
- Ong, C. Y., Weldon, M., Quiring, S., Maxwell, L., Hughes, M., Whelan, C., and Okoniewski, M. (2010). Speed it up. *Microwave Magazine*, 11(2):70–78.
- OpenACC (2013). *The OpenACC Application Programming Interface*. Version 2.0.
- OpenCV (2014). [Opencv.org](http://opencv.org). <http://opencv.org>.
- Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. J. (2007). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113.
- PCI-SIG (2010). *PCIe Base 3.0 Specification*.
- Pellizzoni, R. (2010). *Predictable and Monitored Execution for COTS-based Real-Time Embedded Systems*. PhD thesis, Univ. of Illinois at Urbana Champaign.
- Pellizzoni, R. and Caccamo, M. (2010). Impact of peripheral-processor interference on WCET analysis of real-time embedded systems. *IEEE Transactions on Computers*, 59(3):400–415.
- Pellizzoni, R. and Lipari, G. (2007). Holistic analysis of asynchronous real-time transactions with earliest deadline scheduling. *Journal of Computer and System Sciences*, 73(2):186–206.
- Pieters, B., Hollemeersch, C. F., Lambert, P., and de Walle, R. V. (2009). Motion estimation for H.264/AVC on multiple GPUs using NVIDIA CUDA. In *Applications of Digital Image Processing XXII*, volume 7443, pages 74430X–74430X–12.
- Pilaud, W. (2012). Synthetic aperture radar (SAR) systems for lightweight UAVs enabled by rugged GP-GPUs. *Military Embedded Systems*.
- Rajkumar, R. (1990). Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 116–123.
- Rajkumar, R. (1991). *Synchronization in real-time systems: A Priority Inheritance Approach*. Kluwer Academic Publishers.
- Rajkumar, R., Sha, L., and Lehoczky, J. (1988). Real-time synchronization protocols for multiprocessors. In *Proceedings of the 9th IEEE International Real-Time Systems Symposium*, pages 259–269.
- Rajovic, N., Rico, A., Puzovic, N., Adeniyi-Jones, C., and Ramirez, A. (2014). Tibidabo: Making the case for an ARM-based HPC system. *Future Generation Computer Systems*, 36:322–334.
- Rath, N., Bialek, J., Byrne, P., DeBono, B., Levesque, J., Li, B., Mauel, M., Maurer, D., Navratil, G., and Shiraki, D. (2012). High-speed, multi-input, multi-output control using GPU processing in the HBT-EP tokamak. *Fusion Engineering and Design*, 87(12):1895–1899.



- Renesas (2013). Renesas electronics announces R-Car H2, offering highest CPU and graphics performance for the automotive infotainment market. <http://am.renesas.com/press/news/2013/news20130325.jsp>.
- Roszbach, C. J., Currey, J., Silberstein, M., Ray, B., and Witchel, E. (2011). PTask: operating system abstractions to manage GPUs as compute devices. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 233–248.
- Sandhu, T. (2013). ARM announces Mali-T760 and Mali-T720 GPUs. <http://hexus.net/tech/news/graphics/61733-arm-announces-mali-t760-mali-t720-gpus>. Presented at ARM TechCon 2013 by S. Steele.
- Scicomp Incorporated (2013). *SciFinance: Technical Brief*.
- Segovia, B. and Nanha, Z. (2014). Beignet. <https://01.org/beignet>.
- Seo, Y.-W. and Rajkumar, R. (2014). Detection and tracking of vanishing point on a horizon for automotive applications. In *Proceedings of the 6th Workshop on Planning, Perception and Navigation for Intelligent Vehicles*.
- Sha, L., Rajkumar, R., and Lehoczky, J. (1990). Priority inheritance protocols: An approach to real-time system synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185.
- Shi, L., Chen, H., Sun, J., and Li, K. (2012). vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Transactions on Computers*, 61(6):804–816.
- Shimpi, A. L. (2013a). Intel iris pro 5200 graphics review: Core i7-4950HQ tested. <http://www.anandtech.com/show/6993/intel-iris-pro-5200-graphics-review-core-i74950hq-tested>.
- Shimpi, A. L. (2013b). The iPhone 5s review. <http://www.anandtech.com/show/7335/the-iphone-5s-review>.
- Smith, R. (2013). The AMD radeon R9 290X review. <http://www.anandtech.com/show/7457/the-radeon-r9-290x-review>.
- Smith, R. (2014a). ARM’s mali midgard architecture explored. *AnandTech*.
- Smith, R. (2014b). ARM’s Mali Midgard architecture explored. <http://www.anandtech.com/show/8234/arms-mali-midgard-architecture-explored/8>.
- Steinberg, U., Böttcher, A., and Kauer, B. (2010). Timeslice donation in component-based systems. In *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 16–23.
- Steinberg, U., Wolter, J., and Härtig, H. (2005). Fast component interaction for real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 89–97.
- Suzuki, Y., Kato, S., Yamada, H., and Kono, K. (2014). GPUvm: Why not virtualizing GPUs at the hypervisor? In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 109–120.
- Taymans, W., Baker, S., Wingo, A., Bultje, R., and Kost, S. (2013). *GStreamer Application Development Manual*.

- Texas Instruments (2013). *Multicore DSP+ARM KeyStone II System-on-Chip (SoC)*.
- Urmson, C., Baker, C., Dolan, J., Rybski, P., Salesky, B., Whittaker, W., Ferguson, D., and Darms, M. (2009). Autonomous driving in traffic: Boss and the urban challenge. *AI Magazine*, 30(2):17–28.
- Verner, U., Mendelson, A., and Schuster, A. (2014a). Batch method for efficient resource sharing in real-time multi-GPU systems. In *Distributed Computing and Networking*, volume 8314 of *Lecture Notes in Computer Science*, pages 347–362. Springer Berlin Heidelberg.
- Verner, U., Mendelson, A., and Schuster, A. (2014b). Scheduling periodic real-time communication in multi-GPU systems. In *Proceedings of the 23rd International Conference on Computer Communication and Networks*, pages 1–8.
- Verner, U., Schuster, A., Silberstein, M., and Mendelson, A. (2012). Scheduling processing of real-time data streams on heterogeneous multi-gpu systems. In *Proceedings of the 5th Annual International Systems and Storage Conference*, pages 8:1–8:12.
- Vestal, S. (2007). Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 239–243.
- Vivante (2014). GPGPU. <http://www.vivantecorp.com/index.php/en/technology/gpgpu.html>.
- Ward, B. and Anderson, J. (2013). Fine-grained multiprocessor real-time locking with improved blocking. In *Proceedings of the 21st International Conference on Real-Time and Network Systems*, pages 67–76.
- Ward, B., Elliott, G., and Anderson, J. (2012). Replica-request priority donation: A real-time progress mechanism for global locking protocols. In *Proceedings of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 280–289.
- Watanabe, Y. and Itagaki, T. (2009). Real-time display on fourier domain optical coherence tomography system using a graphics processing unit. *Journal of Biomedical Optics*, 14(6):60506–60506.
- Wei, J., Snider, J., Kim, J., Dolan, J., Rajkumar, R., and Litkouhi, B. (2013). Towards a viable autonomous driving research platform. In *Proceedings of the 2013 IEEE Intelligent Vehicles Symposium*, pages 763–770.
- Wojek, C., Dorko, G., Schulz, A., and Schiele, B. (2008). Sliding-windows for rapid object class localization: A parallel technique. In *Proceedings of the 30th DAGM Symposium on Pattern Recognition*, pages 71–81.
- Xiao, S., Balaji, P., Zhu, Q., Thakur, R., Coghlan, S., Lin, H., Wen, G., Hong, J., and Feng, W. (2012). VOCL: An optimized environment for transparent virtualization of graphics processing units. In *Innovative Parallel Computing (InPar)*, 2012, pages 1–12.
- Zhang, L. and Nevatia, R. (2008). Efficient scan-window based object detection using GPGPU. In *Proceedings of the 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–7.
- Zhang, Y. and West, R. (2006). Process-aware interrupt scheduling and accounting. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 191–201.
- Zhong, J. and He, B. (2014). Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1522–1532.